Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

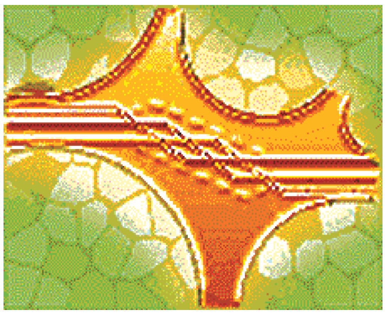
2010

MSc THESIS

Automated Implant-Processor Design

Dhara Dave

Abstract



CE-MS-2010-08

As we move towards an aging population, it is likely that an increasing number of people will require an increasing diversity of implants, but at a lower cost to the society. Also, as computer technology progresses, smaller, more powerful, and less battery intensive implants can be designed. However, present implant design methodology is highly inefficient at meeting these goals as it suffers from non-reuse of existing knowledge by relying heavily on custom designs and ASICs. The SiMS project was started with the goal of creating pre-designed, pre-tested, and pre-certified toolbox of components for biomedical implants that can be assembled in a modular fashion for various application scenarios. One of the most important components in such a tool-box is the processor. Designing such a processor is a non-trivial task and previous work has concentrated on studying the effect of changing the processor input-parameters (such as caches), one parameter at a time. The present work represents a shift in this methodology, as we now allow co-variation in all possible *input* parameters in order to find *optimal* configurations in terms of the *output* objectives - power, performance, and area. Towards this end, we implement ImpEDE – "Implantable-processor Evolutionary Design-space Explorer" – a framework that performs multi-objective optimization of processor parameters, and hence gives as output a Pareto optimal set of processors. The framework consists of a cache simulator and a cycle-accurate processor simulator running benchmarks and workloads designed for medical implants, in order to simulate the optimization objectives. A popular, highly configurable, multi-objective genetic algorithm, NSGA-II, performs the actual optimization. Supporting scripts add modularity by acting as the interface between the genetic algorithm and the simulators, enabling easy replacement with new simulators. The whole framework is parallelized such that extra computation cycles of the idle laboratory CPUs can be utilized, thereby giving a considerable speedup without requiring any special hardware. We perform experiments on the non-dominated solution fronts evolved by the framework on a sub-set of benchmarks, in order to optimize parameters of the genetic algorithm, with an aim towards speeding up convergence. We also examine the effects of changing the workload size run by the benchmarks. A solution Pareto optimal front consisting of optimal processor configurations across all benchmarks is found. This front is used as a reference in order to characterize the benchmarks in the ImpBench suite. Finally, the objective space of the reference front is compared to existing implant designs, and a set of "generic processors" are chosen such that all the existing implant applications studied can be covered.

Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

Automated Implant-Processor Design
An evolutionary multiobjective exploration framework

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Dhara Dave
born in Ahmedabad, India

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Automated Implant-Processor Design

by Dhara Dave

Abstract

As we move towards an aging population, it is likely that an increasing number of people will require an increasing diversity of implants, but at a lower cost to the society. Also, as computer technology progresses, smaller, more powerful, and less battery intensive implants can be designed. However, present implant design methodology is highly inefficient at meeting these goals as it suffers from non-reuse of existing knowledge by relying heavily on custom designs and ASICs. The SiMS project was started with the goal of creating pre-designed, pre-tested, and pre-certified toolbox of components for biomedical implants that can be assembled in a modular fashion for various application scenarios. One of the most important components in such a tool-box is the processor. Designing such a processor is a non-trivial task and previous work has concentrated on studying the effect of changing the processor input-parameters (such as caches), one parameter at a time. The present work represents a shift in this methodology, as we now allow co-variation in all possible *input* parameters in order to find *optimal* configurations in terms of the *output* objectives - power, performance, and area. Towards this end, we implement ImpEDE – "Implantable-processor Evolutionary Design-space Explorer" – a framework that performs multi-objective optimization of processor parameters, and hence gives as output a Pareto optimal set of processors. The framework consists of a cache simulator and a cycle-accurate processor simulator running benchmarks and workloads designed for medical implants, in order to simulate the optimization objectives. A popular, highly configurable, multi-objective genetic algorithm, NSGA-II, performs the actual optimization. Supporting scripts add modularity by acting as the interface between the genetic algorithm and the simulators, enabling easy replacement with new simulators. The whole framework is parallelized such that extra computation cycles of the idle laboratory CPUs can be utilized, thereby giving a considerable speedup without requiring any special hardware. We perform experiments on the non-dominated solution fronts evolved by the framework on a sub-set of benchmarks, in order to optimize parameters of the genetic algorithm, with an aim towards speeding up convergence. We also examine the effects of changing the workload size run by the benchmarks. A solution Pareto optimal front consisting of optimal processor configurations across all benchmarks is found. This front is used as a reference in order to characterize the benchmarks in the ImpBench suite. Finally, the objective space of the reference front is compared to existing implant designs, and a set of "generic processors" are chosen such that all the existing implant applications studied can be covered.

Laboratory	:	Computer Engineering
Codenumber	:	CE-MS-2010-08
Committee Members	:	

For my family

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my advisors, for allowing me to propose and work on this thesis topic, and for the help, support and advice I got from them. Special thanks to my family and friends for all the patience and support. Last but not least I am extremely grateful to our system administrators, Erik de Vries and Eef Hartman for their enormous help with setting up the clusters, and fixing all the (many) problems that arose on the way.

Dhara Dave
Delft, The Netherlands
June 22, 2010

Introduction 1

In 1889, a report was published on how the application of electrical impulses to the human heart could help control its rhythm. It was nearly 70 years later that societal and technological conditions became favourable enough to use this knowledge. The first clinical implant on a human patient was in 1958, Sweden — Arne Larson was given an artificial pacemaker. It failed 2 hours later. A second one failed 2 days later. Arne Larson went on to receive 26 pacemaker implants in his lifetime. However, despite the initial hiccups, artificial pacemakers paved the way for a revolution of sorts in medical technology — implants were here to stay. Since then, many new kinds of implants have been developed. Recently, with declining chip sizes and costs, more and more applications are emerging that could not even be conceived of before.

Today, we are in the midst of another revolution — rapidly advancing technology has changed the way we live and work. One of the driving forces behind this revolution is rapid *adaptability.* As each new discovery or social trend makes older products obsolete, the old adage "*Time is money*" has never been more important. Each company tries to model its practices around frameworks that allow for less waste from previously generated knowledge, leading to quicker *development* and *time to market*; and hence, increased profits. This need for reutilization is a key factor in the drive towards *modularity* — components, both hardware and software, are encouraged to be modular and flexible so that new combinations can be assembled with minimum extra effort.

It is therefore, a little surprising that medical implants have not yet benefited from these insights. As studied by the authors of [71], medical implants are, more often than not, still developed *from scratch.* This leads to longer development times and additional costs. Ballooning health care costs and increasing social debates on medical spending makes this practice quite unacceptable. Needless to say, the faster the new implants come to the market, the more the patients benefit — both, in terms of improved quality of life and reduced cost.

Digital processors lie at the heart of any embedded system designed with modular components. Medical implants, being a special case of embedded systems, are no different. Today's embedded designer has a large arsenal of embedded processors to choose from, each coming with their own trade-offs and constraints in terms of power, performance, and area. While these three objectives are important for any processor, the class of processors for embedded systems exhibit extra constraints.

Medical implants constitute one such highly resource constrained application of embedded systems, and as such, its core component, the processor, is also subject to strict limitations. For example, the implant must be small enough to fit into the implant site. The power dissipation must not be so high that the processor heats up and damages the surrounding tissue. Since frequent surgery to replace batteries would be detrimental to the quality of life of the patient, as well as costly, the system must last long enough to finish its task without running out of power. Most implant applications perform real-time monitoring and/or control, therefore, the implant must be able to handle these without failing. Most importantly, since the application may be life-critical, the system must be extremely fault-tolerant and reliable — in this day and age, receiving 26 implants, as was the case for the first pacemaker, is just not acceptable. Since the processor is a major component of the design, it must also fulfill all these criteria. For these reasons, generic processors available in the market now might not work for implant applications, not least because most of them do not make any claims on reliability.

As we can see, there is a need for processors that can be used specifically for medical implants, and these processors must be generic enough to be used in a wide range of in vivo applications. This work aims to facilitate the design of such processors.

1.1 Context and Motivation

One of the properties needed for modular design is the ability to choose components based on their *black-box* behaviour, rather than implementation details — i.e. as long as the behaviour of the component is accurately specified, one need not worry what goes on inside that component. This way, one can keep a wide-array of components with well-defined characteristics. Whenever there is a need to design an application, components with the desired characteristics can be chosen, and interconnected to each other with minimum knowledge of their internal working. Such design methodology has been used with huge success in modern PC design. A PC designer just needs to know the performance characteristics, power requirements, and pin connections of a modern processor. He generally does not need to know how the ALU is implemented or how the instruction pipeline works. Of course, knowing these can help improve the design choices one makes, but at first, they are not as important as the higher-level black-box parameters. The same principle can be used for embedded design, and consequently, implants design. For the scope of this work, we consider the black-box parameters to be area, power consumption, and performance of the processor – collectively known as the *objectives*. As we shall see, these are not the only design goals that can be added to the system, and other parameters may also be selected in the future. Indeed, it is extremely important to incorporate reliability as a design goal. However, as this is the first stage of implementation, we use a reduced set of design goals, and leave modeling reliability as a goal for future work.

As we shall see, this work was carried out in the context of the SiMS project at the Computer Engineering Department of Delft University of Technology. One of the various deliverables of this project is to propose a generic processor that can be used for medical implants. Designing this processor is not an easy task. It may also be the case that a one-size-fits-all approach may not be feasible given the diversity of implants being designed, and several "generic" processors might be required that among themselves cover a wide range of applications. Therefore, one needs to have a system that can help to choose "good" processor configurations. This system ideally, would map out the entire spectrum of the design space possible, listing the trade-offs between all the objectives. From examining these trade-offs, one can choose configurations that fit the desired design range. A good way for representing the available trade-offs for design is to use the concept of *Pareto Optimality*, which we use in this work.

Obtaining the full trade-offs spectrum as described above however, is a non-trivial task. One must explore all possible configurations of processors, compute the corresponding objectives of area, power etc, and from this, find the *Pareto-dominant* solutions to be included in the final trade-off set. Since typically the design parameters that affect a processor are numerous, computing the behaviour of all possible combinations of these is quite difficult, if not impossible. For example, in this work, we consider 13 processor *design parameters* [Table 3.1] represented by 36 binary bits. If one were to simulate all combinations, one would need to evaluate $2^{36} = 68,719,476,736$ processor configurations! Therefore, we need a way to approximate the trade-off points without performing all possible simulations. Furthermore, even in an approximated scenario, thousands of configurations might have to be considered, evaluated, and compared to get a good approximation of the true *Pareto front*. Therefore, we need an automated method for doing so. Several search and optimization algorithms are available that can be used to perform this very task. In the next sections, we shall see why we chose a particular variant of *genetic algorithms* as the optimization algorithm, and how this algorithm works. We found that the selected algorithm had a very long computation time, and therefore, we describe how we parallelized the algorithm. Finally, we present several trade-off scenarios and the Pareto results the algorithm gives for each of these.

Note that although reliability is one of the major reasons for the need to design processors specifically for implants, the present work does not directly address reliability. Instead, we rely on the idea that processor design can be looked at from a black-box design perspective and provide a flexible and modular framework for doing so. Given such a framework, adding reliability as one of the black-box parameters is more easily done, and is left as future work. For the present work, we concentrate on the following:

1. As we shall see, previous work in the project used certain simulators (XTREM [11] and CACTI [66]). As a continuation of that work, we needed to retain the same simulators. However, future considerations required that this simulator be easily replaceable with another. We did not find an open source/free tool that allowed us to perform multi-objective design space exploration modularly with respect to different processors. Therefore, one deliverable of this work is a free and open design space exploration framework with modular interfaces that enable easy swapping between different processor simulators. We test the modularity of this framework and validate the results to some extent by introducing additional

realtime constraints in the system and observing the trends.

2. In addition to the above problem, the cycle-accurate processor simulators we use also have long simulation times. This necessitates that the DSE algorithm be run in parallel if possible. Further more, we had limited and dynamically changing (hardware) resources available for computation. Therefore, the above mentioned framework is also parallel, scalable, does not need special resources (instead utilizes idle computation cycles of laboratory machines), is easily deployable, and is resistant to dynamic network changes and failures in machines while computing.

3. As further performance optimization of the framework, we also experimented on selecting design parameters (number of generations and crossover probability) for the genetic algorithm in order to speed up its convergence and run-time.This is done by analyzing the distance and spread metrics of the various Pareto fronts found. As far as the authors are aware, such work does not exist in the context of multi-objective genetic algorithms.

4. We use the optimized tool for finding optimal processor configurations. The Pareto front so found is compared to real implant applications in terms of the power, performance and area objectives. We find that our processor configurations are competent in covering the requirements of most of these real applications, and suggest some design points from the front that can be used as starting points for the "generic processors" we require.

5. Finally, we demonstrate the usefulness of the framework in other related contexts by characterizing the benchmarks in the extended ImpBench suite using our DSE framework. This is done as follows: separate Pareto fronts are independantly evolved for each of the benchmarks, which is compared to the front evolved that optimizes a combination of all these benchmarks. Such a study can help processor designers in selecting the benchmarks with appropriate characteristics – for example a single representative benchmark that is closest in behaviour to the combined optimization front can be used instead of the entire suite in case rapid simulations are required. Also, processor designers may opt for a more sensitive benchmark with wide variations in power, performance and area for the allowed range of configurations *or* a benchmark that gives lesser (and hence more predictable) variations in objectives for the same allowed range of processor configurations. As far as we are aware, such a characterization of benchmarks using multi-objective genetic algorithms has not been done.

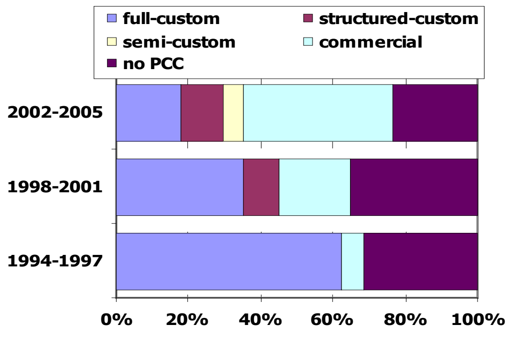
Background and Related Work 2

2.1 The SiMS Project

As stated previously, the practice of redesigning entire implants from scratch is wasteful. The SiMS (Smart Implantable Medical Systems) project [24] at TU Delft aims to address this problem.

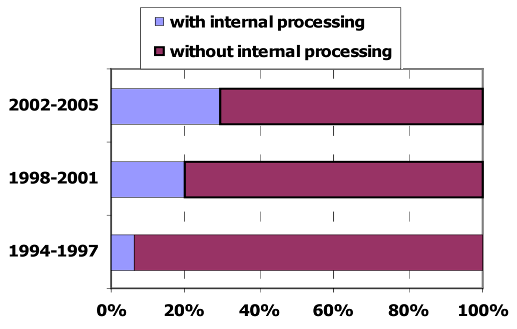
The goal of the SiMS project is to "*deliver a* ***systematic approach*** *(thus, a* ***framework****) that provides medical researchers with a toolbox of ready-to-use, highly reliable sub-systems and models in order to construct (optimal) implants for various medical applications.*" The framework is envisioned to contain hardware and software components that are pre-tested and approved according to the required medical standards. These components can then be mix-and-matched to create different implants catering to different application scenarios. This will significantly reduce design time and costs of medical implants, in addition to providing greater reliability.

Current implant design still relies heavily on custom design as can be seen from Figure 2.1a. However, this needs to change since the goal is towards rapid development of implants using modular hardware components. Additionally, it is typically easier to implement more complex functionality on processors than on designs with no processing cores or on ASICs (Application Specific Integrated circuits) – this is important as the trend is towards implants that require more processing [Figure 2.1b]. Therefore, there is a need for implants processors that are generic enough to cover a wide range of application possibilities. Within the context of the project, Strydis et. al. [74] present the need for such a generic processor and suggest a few characteristics of such a processor. Table 2.1



(a) Use of commercial components is increasing

*PCC : Processing/Controlling Core



(b) More implants require processing capabilities

Figure 2.1: Implant design trends over the years[73]

Feature	**Value**
ISA	32-bit ARMv5TE-compatible
Pipel. depth / Datap. width	7/8-stage, super-pipelined / 32-bit
RF size	16 registers
Issue policy / Instr.window	in-order / single-instruction
I-Cache, L1	64KB, 64-way assoc. (1cc hit/170cc miss)
D-Cache, L1	32KB, 2-way assoc. (1cc hit/170cc miss)
TLB / BTB	1-entry fully-assoc.
Branch Predictor	2-bit Bimodal (32-entry ret. addr. stack)
Write Buffer / Fill Buffer	2-entry / 2-entry
Mem. port no / bus width	1 port / 1 Byte
INT/FP ALUs	1/1
Clock freq. / Implem. tech.	2 MHz / 0.18 m @ 1.5 Volt

Table 2.1: XTREM (Modified) Architecture Details [74]

lists the processor configuration they used in their case study. Furthermore, Strydis et. al. used the XTREM simulator as a base for their work and *independantly* explore the effects of varying processor parameters on processor design in their papers on cache structures[72] and branch prediction [77]. The present work being a logical extension of their work, *co-varies* all such parameters, in order to get optimal configurations across all possible parameter combinations and across all benchmarks; for the three objectives of power, performance and area. Figure 2.2 provides an overview of the processor design methodology followed in the SiMS project, and the context of this work.

2.2 Related Work

The related work of this thesis falls in two major fields - that of implant design, and that of design space exploration. The former, including existing implant design methodologies and the consequent need for generic implants has already been studied by Strydis et.al as part of the SiMS project. Therefore, we shall only briefly review these; the majority of this section is a discussion of the latter topic.

2.2.1 Implant Design

We present a brief overview of a few existing implant designs studied by Strydis [73] in Table 2.2. These designs were chosen from the study as they have listed power, performance and area characteristics; and therefore, can be compared with the processor configurations we derive from our exploration. This comparison is presented in Chapter 4, Section 4.7.

The performance metric listed in these applications is the worst case time to execute the (repetitive) task they are designed for. However, these tasks are highly application dependant and may vary widely. One way of ensuring a fair comparison is to normalize

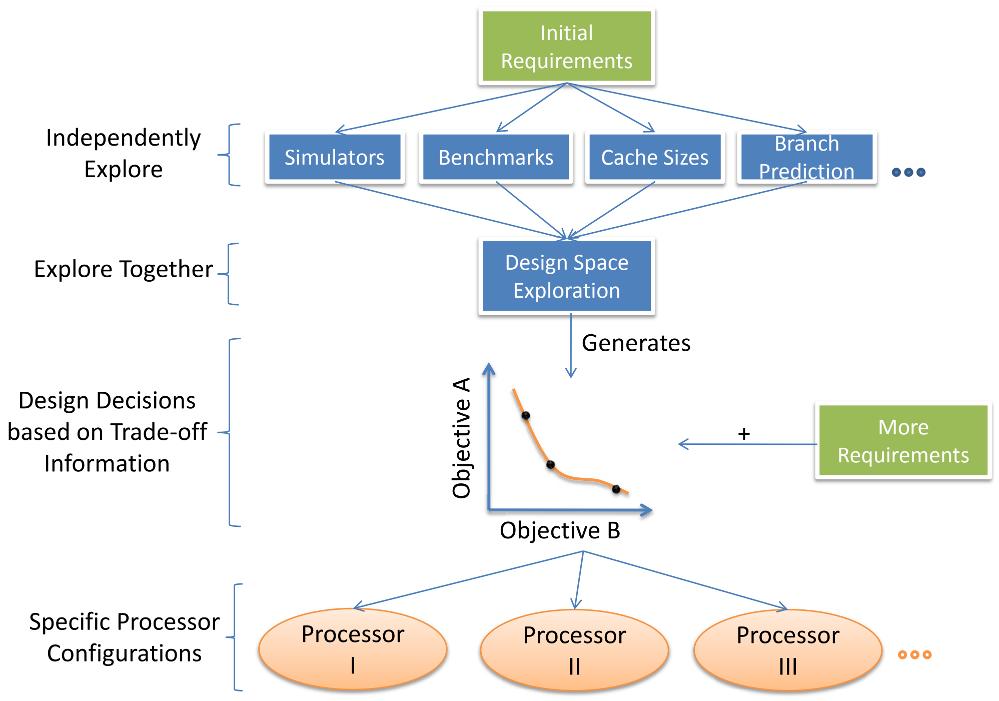


Figure 2.2: Overview of the SiMS processor design methodology. Independant exploration has already been done by Strydis[72, 77], we focus on the other steps

the amount of data processed in a single iteration of the task. We note that each of these applications are simple stimulation or measurement tasks, with each loop simply processing the ADC data. Therefore, in order to normalize, we assume each loop reads 10KB ADC data at a time (possibly buffered). This approximation is the *best case* scenario for these applications as it assumes there is no (a) data-processing time or (b) buffering times. On the other hand, as we shall see, the performance listed for our processors is also for 10KB data, but the tasks run are the ImpBench benchmarks, which have non-trivial processing times. Therefore, with these assumptions, the real applications represent the *worst case* scenarios for our processors.

2.2.2 Processor Design

For an in-depth analysis of designing processors from design space exploration to synthesis, the interested reader can refer to Gries[31], who contends that the two independant problems in design space exploration (DSE) are how to evaluate a single design point, and, how to cover the design space; and goes on to discuss both in detail. In the context of this work, the first of the problems is not an issue, as previous work in the project has already established the XTREM and CACTI simulators (refer Section 2.3.6). However, the question of how to cover the design space is important. The design space for a processor is huge, and while we would like to cover as much of it as possible, evaluating the space for every single processor configuration possible is virtually impossible. Many general techniques have been proposed in literature that

#	Author	Year	Application	Functionality	Chipset Area (mm^2)	Peak Power (mW)	Worst-case Single loop Execution Time (msec)	ADC resolution (bits)	Worst-case Single loop Execution Time (Normalized) (sec)
(a)	Smith et al. [67, 59, 58]	1998	restoration of paralyzed-muscle functionality, MES	stimulation/ measurement	937.50	96.00	10.00	12	68.27
(b)	Eggers et al. [23, 22, 38]	2000	ICP-based diagnosis for brain diseases	measurement	58.50	0.24	10.00	10	81.92
(c)	Rollins et al. [63]	2000	continuous ECG for understanding spontaneous cardiac arrhythmias	measurement	4209.67	34.00	1.00	12	6.83
(d)	Valdastri et al. [84]	2004	gastric-pressure monitoring	measurement	162.00	50.40	0.04	10	0.33
(e)	Au-Yeung et al. [3]	2004	continuous monitoring of AEG, delivery of atrial ATP	stimulation/ measurement	5106.00	115.30	3.00	10	24.60
(f)	Liang et al. [49]	2005	ENG	measurement	1350.00	90.00	0.09	10	0.75

Table 2.2: Overview of a few existing implant applications

explore the design space and search for optimal points. In Section 2.3.1 we take an in depth look at the possible generic optimization techniques, and the reasons for selecting a multi-objective genetic algorithm for this work. In this section, we shall briefly list some tools and techniques developed specifically for design space exploration of processors.

Hekstra et al. [35] explored the TriMedia CPU64 design using pruning – they first probe the design space in order to identify the architectural parameters that affect overall performance the most. The extreme values of these parameters provide 'corner cases', and help in bounding the space to be explored in detail.

DESERT [81] (DEsign Space ExploRation Tool) is a meta-programmable tool for pruning large design spaces using constraints. It represents the design space as a generic structure based on alternatives and parameters, and therefore can be used for diverse applications. Mohanty uses the MILAN [53] (Model based Integrated simuLAtioN) tool, based on DESERT, for pruning design spaces of heterogeneous multi-core systems. The authors of the Artemis (Architectures and Methods for Embedded Media Systems) [57] also work on exploration of heterogenous multi-core environments, but focus more on modeling and simulation than techniques for DSE.

Cho et. al [9] content that micro-architecture design is better done by considering dynamic behaviour of workloads rather than designing for worst-case workload behaviour. They use wavelet-based multi-resolution decomposition and neural network based non-linear regression modeling to reason about workload dynamics (in terms of performance, power, and reliability) across the micro-architecture design space.

The PICO [43] framework designed at HP Labs, given C code, outputs application specific embedded computer systems optimized for cost vs. performance; where the 'computer system' consists of a EPIC/ VLIW (explicitly parallel instruction computing/very long instruction word) and an NPA (non-programmable accelerator). It uses a *space walker* – which may be a heuristic, or brute force depending on the search space – to search the design space. A 'component assembler' outputs HDL code for the processors specified by the space walker by assembling low-level components from their *component library.*

Xie et. al [88] provide an in depth discussion on design space exploration for 3D Integrated circuits, including CAD and design tools, and simulators. They use simulated annealing to automatically find floorplans for the ICs.

Stijn et. al [25] evaluate various automated single and multi-objective optimizations for exploring high performance embedded out-of-order processor designs. They found that a genetic local search algorithm outperforms all other techniques for their application.

Ascia, Catania and Palesi [2] propose using genetic algorithms to perform design space exploration in processors. They apply a genetic algorithm to optimize the memory hierarchy in terms of area, power and mean access time. However, they use a single-objective genetic algorithm and model the fitness function as a product of the three objectives. As we shall see in Section 2.3.2, such techniques face drawbacks that reduces their suitability for use with design spaces whose shapes are not known in advance[1], as in our case.

Thiele et. al. [83] present domain specific design space exploration for network processor architectures. They specify models for packet specific tasks and network traffic ("encoding"); methods to estimate delays and queuing memory ("simulation"); and use an evolutionary algorithm to perform multi-objective design space exploration ("optimization"). This work is perhaps the closest to our work. However, to the best of our knowledge, design space exploration with respect to implantable systems has not been previously studied.

2.3 Background

In this section, we formally describe the terminology used in the paper, and the two major categories of optimization - single-objective and multi-objective, and their applicability to our case. We then give a brief overview of various optimization heuristics, why we

[1]Since the design space may well be non-convex, for which this method does not work

chose genetic algorithms as our heuristic of choice, and describe terminology associated with genetic algorithms. Finally, we give an overview of the simulators (XTREM and CACTI) and benchmark suite we employ (ImpBench).

2.3.1 Notes on Optimization

Optimization is defined as "*an act, process, or methodology of making something (as a design, system, or decision) as fully perfect, functional, or effective as possible; specifically: the mathematical procedures (as finding the maximum of a function) involved in this*" [20]. In processor design, it would mean the process of finding as perfect a processor as possible; perfect being subject to the design objectives we would like to characterize the processor with.

Formally, an optimization problem is defined as [18, eqn. 2.1]:

$$\begin{array}{lll} \text{Maximinize/Minimize} & f_m(x), & m = 1,2,...,M; \\ \text{subject to} & g_j(x) \geq 0, & j = 1,2,...,J; \\ & h_k(x) = 0, & k = 1,2,..,K; \\ & x_i^{(L)} \geq x_i \geq x_i^{(U)}, & i = 1,2,...,n. \end{array}$$

A feasible solution to this problem will consist of an n-dimensional vector $x = (x_1, x_2, ..., x_n)^T$ that falls within the lower and upper variable bounds ($x_i^{(L)}$ and $x_i^{(U)}$ respectively) and satisfies the $j+k$ constraints. The n-dimensional region where the feasible solutions lie is known as the search space. We can evaluate each of the m objective functions on every point x in the search space. The resulting m dimensional space is known as the *objective space* or *design space* (Refer: Figure 2.3). In this work, we shall mostly refer to it by the latter name.

2.3.1.1 Pareto optimality

Most classical optimization techniques focus on optimizing a single variable or multiple variables with respect to a single objective. However, many real engineering problems involve more than one objective. Furthermore, for multiple-objective problems, the objectives are often conflicting, such as minimizing chip-area while maximizing performance. In such a problem, there is no single optimal solution. Instead, there exist a number of solutions that are all optimal. In the absence of more case-specific information (which may be purely qualitative, or even be absent altogether), there is no way of saying which solution from this set of solutions is **the** optimal. In such cases, the decision makers can be provided with the entire set of optimal solutions and the trade-offs they represent. They can then make decisions based on these trade-offs and the particular manifestation of the problem, or, when more information is available. The most interesting set of trade-offs for a given problem is the set of *Pareto points* (Figure 2.4). A point in the design space is a Pareto point if there is no objective that *dominates* it. In this context, *dominance* is a strict partial order relation defined as (adapted from Deb [18], Coello[10]):

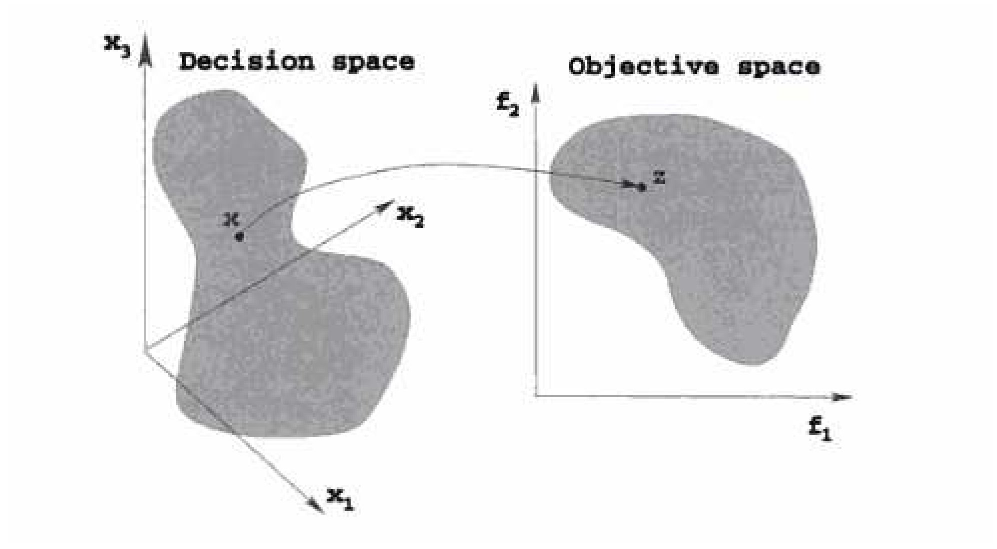


Figure 2.3: Representation of the decision variable space and the corresponding objective space [18, Figure 5, Page 15]

$$\text{Let} \quad \left.\begin{aligned} \vec{u} &= (u_1, ..., u_m) &= (f_1(x_1), ..., f_m(x_1)) \\ \vec{v} &= (v_1, ..., v_m) &= (f_1(x_2), ..., f_m(x_2)) \end{aligned}\right\} \text{where } f_i = i^{th} \text{ objective function}$$

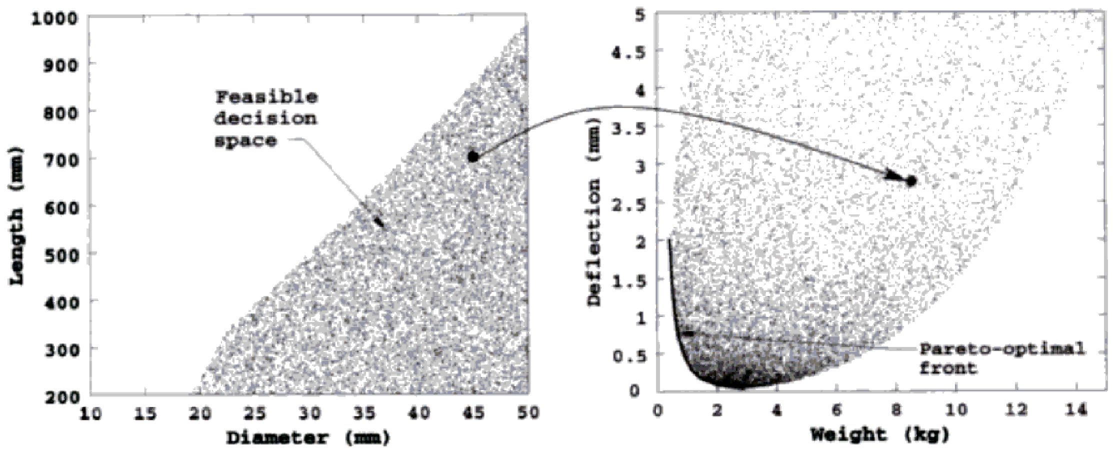
Solution x_2 **dominates** solution x_1 *if and only if*:

$$\begin{aligned} & u_i \le v_i, && \forall i \in \{1, 2, ..., m\} \\ \text{And,} \quad & u_i < v_i, && \exists i \in \{1, 2, ..., m\} \end{aligned}$$

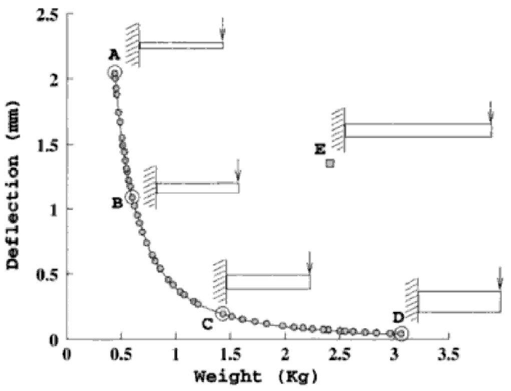
In words, this implies: the solution x_2 is said to dominate solution x_1 if x_2 is no worse than x_1 in all objectives, and, x_2 is strictly better than x_1 in at least one objective.

For example, in Figure 2.4b, E is not a Pareto-point since it is dominated by other points - there exist points which have a lower weight given the same deflection as E, and points that have a lower deflection given the same weight as point E. The Pareto points are A,B,C,D because they are non-dominated in the entire feasible objective space. The entire set of such Pareto points is known as the *Pareto front*.

In the context of this work, having the Pareto front of the problem available means that the decision makers can be given the whole spectrum of trade-offs pertaining to power, performance, and area. Since the aim is to have generic processors, they can then choose a few of these processor configurations that between themselves are representative of most of the medical implant applications.



(a) Feasible Decision Space and the Corresponding objective space in a real engineering problem (deflection and weight depend on length and diameter of a beam)



(b) The pareto front corresponding to the above problem. A,B,C,D are Pareto points and form the Pareto-front; E is dominated by other points and hence is not a Pareto-optimal point.

Figure 2.4: It is easier to choose design parameters (length, weight) given the Pareto front of the objectives (deflection, weight) [18]

2.3.2 Single vs. Multi-optimization

The classical single-objective optimization methods can be used to perform multi-objective optimization by reformulating the multi-objective optimization problem into a single-objective one. This can either be done by combining the objectives into a single aggregate objective [44], or by only considering one of the objectives and moving the rest to a constraint set [51]. The first approach can be implemented, for example, with finding the weighted sum of each of the objectives - instead of objectives X and Y, we optimize $w_1X + w_2Y$, where $w1$ and $w2$ are appropriate weights. However, this is difficult, because the objectives may be non-commensurable, their relative importance may not be quantifiable, and, the range of values they can take may not be known in advance. This makes it difficult to decide the weights by which each variable must be scaled, since even slight changes in the weights can lead to quite different solutions [46]. The second approach also faces almost the same drawback - to reformulate the objectives as constraints, the range of values of these former objectives must be known in advance. In spite of these limitations, these methods are attractive because they are simpler to understand and implement. In the first stage of implementation, we used a single-objective Genetic Algorithms that used the weighted-sums approach for finding the fitness of an individual. However, this was quickly abandoned as we could not logically assign the values of the weights since there was no rationalization for choosing one objective relative to the other without more information about the problem domain. Also, indeed we did not have the absolute upper and lower limits of the three objectives (power, performance, area).

The above methods provide the designer with only a single solution. By changing the aggregation method slightly, one can reduce the impact of the weight-selection process and, instead of a single solution, provide the designer with a Pareto-front solution. In case of the weighted-sums-approach, this can be done by iteratively changing the relative weights of the objectives and optimizing each of these sums. For example, for objectives X and Y, we optimize $X + w_1Y, X + w_2Y ... X + w_nY$. This gives us n tradeoff points between X and Y depending on how much importance we attach to Y relative to X. However, this approach has its own drawbacks such as inability to ensure a uniformly distributed set of Pareto-optimal solutions, difficulty with problems having a non-convex objective space, and no guarantees on optimality beyond satisfying the first order optimality criteria [18, Chapter 3] [14]. The other classical methods "fixed up" for multi-objective optimization all face similar drawbacks, summarized in Table 2.3.

Therefore, we need a technique specifically developed for multi-optimization that does not face the above drawbacks. In the next section, we introduce NSGA-II, a Genetic Algorithm that does exactly this.

2.3.3 Heuristic Algorithms

"*Trying to find the global optimum of a multi-objective problem is an NP-Complete problem*"[10]. Therefore, the best we can do is use a heuristic to approximate the solution space. In this section, we list some of the popular meta-heuristic techniques that can be applied to any search and optimization problem.

Random Optimization[68] At each iteration, the design space is sampled randomly, and the design point is evaluated with respect to the optimization objective. The best design point seen so far is saved, and the candidate solution point at the end of a fixed number of iterations is considered the final solution. The idea is that if the iterations are allowed to continue for a long enough time, the algorithm will eventually encounter the optimal solution. This approach does not have problems with local minima. However, convergence depends on the kind of design space and the quality of solution required – if there is exactly one optimal solution, it may not be found; on the other hand if there are several design points each having an optimal value, then there is better likelihood of finding one of them.

Table 2.3: Summary of methods using single objective optimization techniques to perform multi-objective optimization

Method	**Description**	**Advantages**	**Disadvantages**	**Formula**	**Ref.**
Weighted Sum	Select a weight w_m for each objective function $f_m(x)$, minimize the weighted sum	Simple, Intuitive, Guaranteed for Convex problems	Adding non-commensurable Objectives, Hard to get uniform Pareto-point spread, Doesn't work for non-convex problems	Minimize $F(x) = \sum_{m=1}^{M} w_m f_m(x)$ subject to $g_j(x) \geq 0, \quad j = 1, ..., J;$ $h_k(x) = 0, \quad k = 1, ..., K;$ $x_i^{(L)} \leq x_i \leq x_i^{(U)}, \ i = 1, ..., n;$	[14]
ϵ-Constraint	Optimize one objective $f_\mu(x)$, reformulate other objectives $f_m(x)$ $(m \neq \mu)$ as constraints	Works for convex and non-convex problems	If wrong value of ϵ is selected, may not give any solution. More the objectives, more ϵ_s need to be chosen from knowledge of problem domain	Minimize $f_\mu(x),$ subject to $f_m(x) \leq \epsilon_m, \quad m = 1, ..., M, \ m \neq \mu;$ $g_j(x) \geq 0, \quad j = 1, ..., J;$ $h_k(x) = 0, \quad k = 1, ..., K;$ $x_i^{(L)} \leq x_i \leq x_i^{(U)}, \ i = 1, ..., n;$	[33]
Weighted Matrix Method	Generic case of Weighted Sums method. Minimize distance measure l_p of any solution x from the ideal solution z^*. Depending on value of p chosen, minimize weighted sum, Euclidean distance or highest deviation (for large p)	Guranteed to find each Pareto-optimal solution	Needs normalization of objectives - Bounds of objectives must be known, Need to independantly optimize each of M objectives before problem is optmized (for getting z^*)	Minimize $l_p(x) = \left(\sum_{m=1}^{M} w_m \lvert f_m(x) - {z_m}^* \rvert^p \right)^{1/p}$ subject to $g_j(x) \geq 0, \quad j = 1, ..., J;$ $h_k(x) = 0, \quad k = 1, ..., K;$ $x_i^{(L)} \leq x_i \leq x_i^{(U)}, \ i = 1, ..., n;$	[18, Chapter 3]
Benson's Method	Similar to Weighted Matrix method, except the reference solution z^0 is a randomly chosen feasible non-Pareto solution.	If appropriate z^0 is chosen, can find solutions in the non-convex Pareto optimal region	Additional constraints introduced for restricting solution in region dominating z^0, non-differentiable objective function	Maximize $\sum_{m=1}^{M} max(0, ({z_m}^0 - f_m(x))),$ subject to $f_m(x) \leq {z_m}^0 \quad m = 1, ..., M;$ $g_j(x) \geq 0, \quad j = 1, ..., J;$ $h_k(x) \geq 0, \quad k = 1, ..., K;$ $x_i^{(L)} \leq x_i \leq x_i^{(U)}, \quad i = 1, ..., n;$	[6]
Value Function Method	Also called utility function method. The user provides a value function U relating all objectives, this function is maximized	Simple idea, if value function information is available	Solution depends on value function chosen, Need a value function applicable over the entire search space	Maximize $U(F(x))), \quad F(x) = (f_1(x), ..., f_M(x))^T$ subject to $g_j(x) \geq 0, \quad j = 1, ..., J;$ $h_k(x) \geq 0, \quad k = 1, ..., K;$ $x_i^{(L)} \leq x_i \leq x_i^{(U)}, \ i = 1, ..., n;$	[26]

Hill Climbing[64] As a kind of greedy algorithm, it makes locally optimal choices at each iteration in an attempt to find the global optima. The algorithm starts with a random initial solution to the problem at hand. The solution is then mutated, and if the mutation results in a better solution than the previous one, the new solution is kept; otherwise, the current solution is retained. The algorithm is repeated until no mutation can be found that causes a more optimal solution than the current solution, and the current solution is returned as the result. The drawback is that often, the algorithm gets stuck at local optima and can not find the global optimum – although methods have been suggested to counter this. Also, for a noisy design space, there may be problems in converging.

Simulated Anealing[45] Inspired by the annealing process in mettalurgy, each step of the algorithm replaces the current solution by a random "nearby" solution. The nearby solution is chosen with a probability that depends on the difference between the corresponding function values of the current and nearby solution, as well as a global parameter T (called temperature), that is gradually decreased during the process. This dependency is such that the current solution changes almost randomly when T is large, but increasingly slowly towards (local) optima as T goes to zero. As was the case of hill climbing, this method may also get stuck in local optima, but with a lesser likelihood of doing so, due to the initial wide range of searches.

Tabu Search [27] This algorithm is similar to the hill climbing approach, except that the algorithm keeps track of solutions already examined through memory structures. The solutions recently examined and discarded are marked as 'taboo' and therefore can not be re-visited. Therefore, the search performs faster by avoiding unnecessary re-evaluations. Since it is so similar to the hill climber, it faces the same drawbacks.

Ant Colony Optimization[21] The algorithm searches for an optimal path in a graph; based on the behavior of ants seeking a path between their colony and a source of food. Ants wander from the colony to random places in search for food and then return to the colony. On the path they take to return, they deposit a trail of pheromones. If an ant travelling back encounters a pheromone trail, it follows this trail instead of a new path, thereby depositing more pheromones on it and strengthening the path. As time passes, trails also lose pheromone. This processes of addition and removal of pheromones eventually leaves behind optimal trails. This behaviour is extended to the design space to find optimal solutions – ant colony algorithms are regarded as populated metaheuristics with each solution represented by an ant moving in the search space. Ants mark the best solutions and take account of previous markings to optimize the search. The drawback is that real life problems are sometimes difficult to formulate as equivalent ant-colony problems.

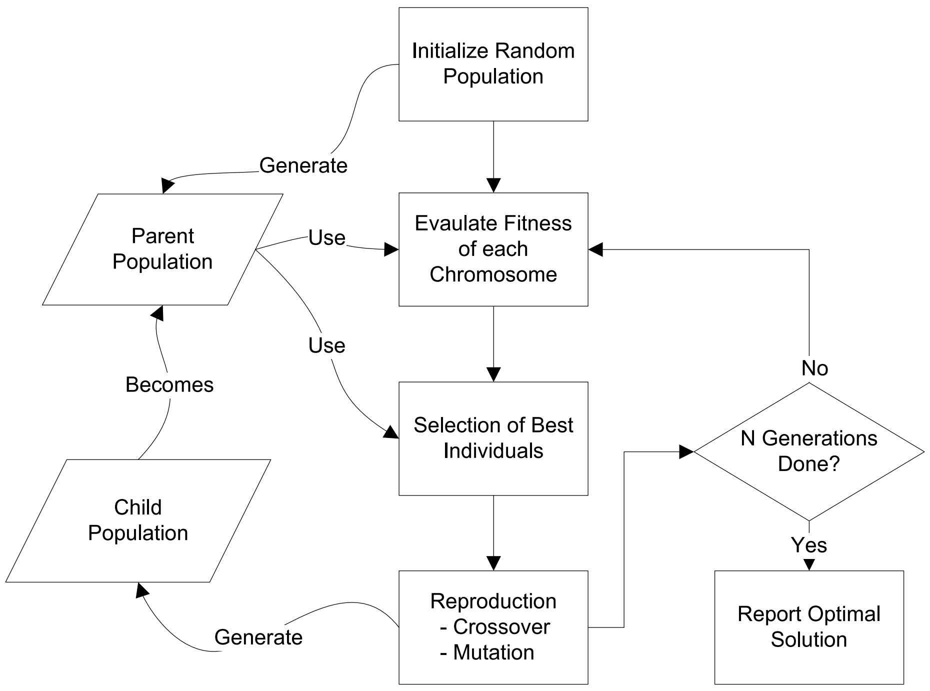
Evolutionary Algorithms[28] These algorithms solve the problem of convergence local optima, are widely studied and applied, and, in our opinion, easier to formulate as a search problem for our application. Detailed discussion on this algorithm can be found in Section 2.3.4.

2.3.4 Genetic Algorithms

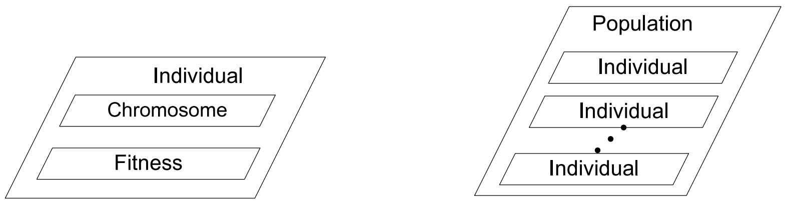
Evolutionary Algorithms are a class of Artificial Intelligence algorithms that mimic Darwin's theory of natural selection to solve search and optimization problems. Genetic Algorithms (GAs) are a special class of Evolutionary Algorithm, defined by their pioneers [28] as "*probabilistic search procedures designed to work on large spaces involving states that can be represented by strings.*" Simple as this may sound; Genetic Algorithms are extremely powerful and can be applied without much modification to a large class of problems. They overcome many of the problems associated with other search techniques, such as — the convergence does not depend heavily on the chosen initial solution; the algorithm has mechanisms to not get stuck at local-optima; discrete, noisy, and dynamic search spaces are efficiently handled; and the algorithm can easily be parallelized [18, Page 82].

The basic Genetic Algorithm encodes the input variables as a string of bits, known as a *chromosome*. A chromosome belongs to an *individual*, which represents one particular solution point in the search domain. A collection of these individuals forms a *population*. Therefore, the population represents the mutually independent solutions under consideration. The algorithm tries to *evolve* the population in such a manner that the *fitness* of the population minimizes (or maximizes). The fitness here is represented by a problem-specific *fitness function*, and quantitatively represents the search or optimization objective. Each evolutionary step of the algorithm creates a new (*child*) population from the previous step's (*parent*) population by selectively choosing fit individuals to be propagated into the next generation. These individuals are then subject to change via *cross over* (Chapter 3, Figure 3.3) – exchange of genetic material – and *mutation* (Chapter 3, Figure 3.2) – destruction of some encoded information and introduction of new, random information – in order to drive the search into new and previously unexplored directions [86]. Some genetic algorithms ensure that the best solutions found so far are preserved. This approach, known as *elitism*, helps ensure that the random processes do not destroy good solutions; the algorithm always moves towards a better solution, never leading to a worse one. The basic GA steps and components are shown in Figure 2.5.

The definitions above are for a single objective genetic algorithm. In case of multi-objective GAs, the population represents the candidate solutions that lie on the (approximate) Pareto front. The algorithm in this case then tries to evolve the entire solution front towards the true (but unknown) Pareto front. The selection procedure in this case becomes slightly different than that in the single objective version. In the single objective algorithm, deciding which individuals from a population get to reproduce and move on to the subsequent generations is done by sorting the individuals according to their fitness - in the simplest implementation, the top few individuals from the sorted list get through to the next generations, others are



(a) Steps of a Genetic Algorithm



(b) An Individual consists of a Chromosome (encoded attributes) and its fitness (quality w.r.t. The objectives)

(c) A Population consists of N Individuals

Figure 2.5: Schematic and components of a Genetic Algorithm

discarded; more complicated strategies use a probabilistic selection approach on the selected individuals. Since now we want to collectively better the entire population, simple sorting will not work. We need a mechanism that ensures that the front found at every generation moves forward. The algorithm that we use for this work, called NSGA-II, uses 'non-dominated sorting' specially designed for multi-objective GAs to accomplish this. The interested reader can refer to the detailed working of NSGA-II in [19].

We chose NSAGA-II as it is state of the art, and is one of the most widely used and studied multi-objective algorithms in literature, along with its competitor, SPEA2 [90]. Moreover, the NSGA-II sources are freely available from the authors. We use a modified version of this original source code in our work.

benchmark	**name**	**size (KB)**	**dyn. instr.* (average) (#)**	**dyn. μops* (average) (#)**
Compression	miniLZO	16.30	233186	323633
	Finnish	10.40	908380	2208197
Encryption	MISTY1	18.80	1267162	2086681
	RC6	11.40	863348	1272845
Data integrity	checksum	9.40	62560	86211
	CRC32	9.30	418598	918872
Real applications	motion	9.44	3038032	4753084
	DMU4	19.50	36808080	43186673
	DMU3	19.59	75344906	107301464
Stressmarks	stressmotion	9.40	288745	455855
	stressDMU3	19.52	124212	224791

Table 2.4: ImpBench benchmarks. (*) indicates typical values for $10 - KB$ workloads, except for DMU-variants which use their own special workloads. [79]

2.3.5 Benchmarks and Workloads

The performance of a processor depends not only on its internal components but also on the applications that run on it. Therefore, to get an accurate representation of the suitability of a generic implant processor, we need to run a set of benchmarks to characterize it that are representative of the application domain. The ImpBench benchmark suite [78] is proposed for this very purpose, we use an extended version of the ImpBench suite[79]. The applications included in the suit are classified as *lossless compression*, *symmetric-key encryption*, *data integrity*, and *synthetic programs* that are representative of actual applications. A brief overview of the benchmarks can be found in Figure 2.4. We use these benchmarks for evaluating the fitness of the individuals in our Genetic Algorithm.

The behaviour of programs also depend on the inputs given to the program. To get a true representation of a processor, we must therefore also choose a suitable workload and workload size for running the benchmarks. For profiling the benchmarks in the ImpBench suit, 7 representative workloads have been selected based on real data. The different benchmarks perform differently with respect to these workloads - the “motion” application and the compression algorithms depend heavily on both the type of input data and the size of the input data [75], encryption and data-integrity algorithms only depend on the input data size [76], the DMU application is a tight kernel reading its own fixed input values and hence does not depend on the either the input size or the input data. All benchmarks (except DMU) show worst case behavior (in terms of IPC) with the EMGII input type[61]. Therefore, we use EMGII as the workload for all our experiments. We also selected the higher available workload size of 10KB. In Chapter 4 we will study the effect of changing this workload size to the lower size of 1KB.

2.3.6 Simulators

Ideally, to characterize a given processor configuration, one would make an HDL implementation of the processors, synthesize this, and thus compute its power, performance and area requirements. However, we need to implement thousands of processor configurations to feed the Genetic Algorithm (GA) in its exploration of the design space. This process is cumbersome if done manually, and difficult to automate. Therefore, we need simpler simulators to approximate the behaviour of the processors. These simulators model processor behaviour as function evaluations. Needless to say, this will introduce approximations into the values found. A designer requiring a certain processor can select promising candidates from the design space found by the GA while keeping this approximation in mind.

The primary simulator we use is XTREM [11], a modified version of SimpleScalar [4]. The XTREM simulator is a cycle-accurate, microarchitectural, power- and performance-functional simulator for the Intel XScale core [74], with an average performance error of only 6.5% and an average power error of only 4% [11]. Moreover, the simulator is open, free, and readily available.

XTREM has a 32-bit ARMv5TE-compatible ISA with a 7/8-stage pipeline, a 32-bit datapath, 16 registers, in-order instruction execution, and 2 MHz clock frequency. It supports modelling of, among other things, L1 and L2 caches, different branch predictor schemes, ALUs, write and fill buffers, memory ports, and bus width. It was chosen because of its precision, ease of use, availability, and as mentioned before, because previous work in the project has been done using this simulator. However, while using, we did find flaws in the simulator. Therefore, the future work on the project should replace this simulator.

The second simulator we use is the popular CACTI (Cache Access Latency and Power Estimation Tool), which is "*An integrated cache and memory access time, cycle time, area, leakage, and dynamic power model*" [47]. Specifically, we use CACTI 3.0, which includes modelling support for the area of caches, caches with independently addressed banks, fully-associative caches, a power model, technology scaling, among other things. We use CACTI for modeling the areas of the caches and the branch prediction table, which are used to approximate the total processor area. This is justified as the major component of the chip area are cache structures [2]rather than processing blocks, with the processing block size hardly changing from design to design [3]. In fact, according to [70], caches take caches consume approximately 90% of transistor count, taking up 60% of the area. Also, as we shall see, the way our design works, there is minimal change in the processing blocks, but we do vary the cache configurations. Therefore, modeling the area through the cache structures gives a good relative metric to comparison area.

[1]For the processor modeled by XTREM, core size is 0.13765 cm^2 whereas the cache size is about 0.20647 cm^2[70]

[2]The size of the ARM cores hardly changed from ARM2 (30,000 transistors) to ARM6 (35,000 transistors), whereas the cache sizes changed from no cache to 4KB cache. The total transistor count changed from 30,000 in ARM2 to 360,000 transistors in ARM6(including MMU, caches, write back buffer)[12].

Experimental Setup 3

As stated previously, there is a need for automating the optimization process that searches for good processor configurations. In addition to this, we also need a way to automate the process of creating different processor configurations and evaluating them. We delegate this task to two simulators, XTREM [11] and CACTI [66]. The first takes into account all 13 of our processor input parameters and simulates the performance and power requirements of the processor so formed, while we use the later to approximate the area of the processor.

We also note that the performance, and power consumed by a processor depend heavily on the kind of application run on the processor. Many applications in turn, themselves depend on the data input to these applications. Therefore, we also need to select carefully what applications and data-sets we run on the processor to gather its power and performance metrics.

Therefore, our basic experimental setup is as follows: A genetic algorithm (GA) evolves towards finding the Pareto-front of the 3-D space consisting of processor power, performance and area. To simulate the power, performance and area behavior of the processors the GA considers, we use the XTREM and CACTI simulators. Since the output of the XTREM simulator depends on the program run on that instance of the processor, we run several benchmarks to evaluate the processor. These benchmarks themselves depend on the input given to the benchmarks. Therefore, we run the worst-case input data-sets on the benchmarks.

We found the above setup to be extremely slow and time-consuming. Therefore, we parallelized the setup so that the expensive computations could run on several computers at a time. In this way, we utilized the idle CPU times of available laboratory computers to run our simulations faster. The final experimental setup is shown in Figure 3.1. Note that the entire experiment was run on the Linux-based operating system Fedora 8.

3.1 Variables Chosen

The processor parameters we chose to include in the search space depended both on what we wanted out of the processor and the capabilities and limitations of the simulators we had. The same can be said about the ranges of these parameters; for most of them, we used the maximum allowed range that the simulators could support, so that we could capture as much of the design spectrum as possible. These variables are summarized in Table 3.1

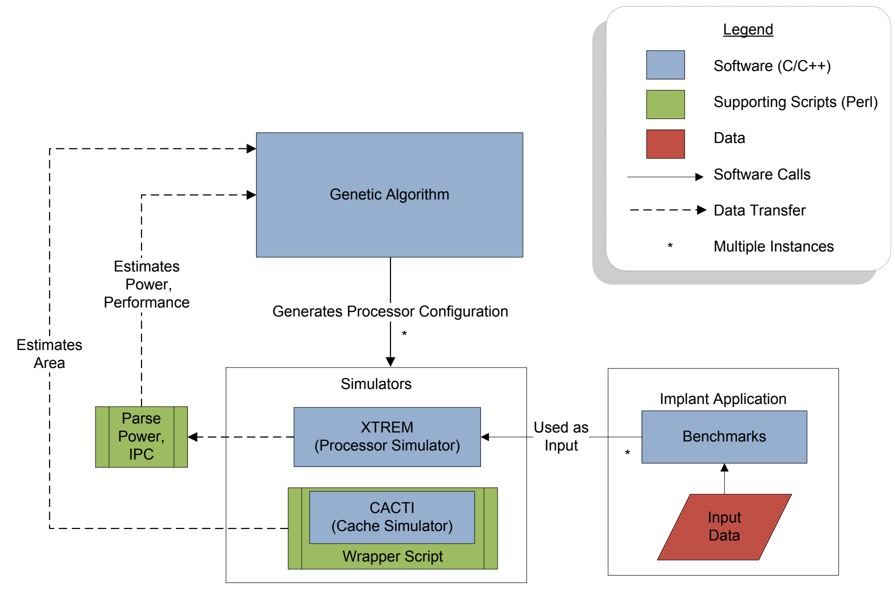


Figure 3.1: Experimental Setup

3.1.1 Clock Frequency

The clock frequency of a processor affects both its power consumption and performance. However, we found that although the XTREM simulator accepted frequency as an input, running the simulator with different clock frequency values did not affect the results. We tried debugging XTREM for this, but this was later abandoned due to time constraints. In the future, it would be interesting to see the effect of clock frequency on processor design. For the purpose of this work, XTREM runs on a default clock frequency of 2 MHz.

3.1.2 Branch Prediction

For more than two decades now, all processors, including embedded processors have used pipelining to improve performance [36, Chapter 3]. Pipelining overlaps the execution of multiple instructions and thereby takes advantage of Instruction Level Parallelism (ILP) that exists among the actions needed to execute an instruction. However, in real programs, instructions are not completely independent of each other. Dependencies in data means that the pipeline needs to stall for the results of one instruction before it can proceed with the next. Branches introduce one class of these dependencies, known as control dependencies. The stalls due to these dependencies can significantly affect performance, given that branches on an average make 12% of the total instruction executed [77, Table II (Derived from)].

To reduce the number of stalls from conditional branch instructions, *speculative execution* takes place - the hardware tries to guess whether the branch will be taken, and executes the next instructions before the result of the branch instruction is known. If its prediction is accurate, there is no loss in performance because of the branch. On the other hand, for a failed prediction, some penalty is incurred because the system must revert to the pre-speculative execution state and execute instructions starting at the correct branch.

There are two kinds of branch prediction schemes, *static* and *dynamic*. In static schemes, "*The action taken does not depend on the dynamic behaviour of the branch*" [36, Chapter 3]. Two kinds of such schemes are *predicted-not-taken* and *predicted-taken*. Static schemes have a simpler implementation, and therefore less hardware (area) requirement. We model both these static schemes in our design. On the other hand, using dynamic schemes may help in improving performance more by giving a better estimation of the outcome of the branch. Dynamic prediction schemes use a buffer to store past branch states and based on that, try and predict the current branch. While research in branch prediction has been extensive, most of the solutions target high-performance processors, not processor meant for embedded applications. Moreover, most embedded CPUs do not make use of branch predictor units due to low battery and/or power considerations. For the SiMS processor, we did not want to dismiss branch predictors all together without some actual data, yet, extremely complex schemes would be quite unrealistic for the application. Therefore, to strike a balance between performance and complexity, out of the many dynamic prediction schemes available, such as correlating predictors, tournament predictors, even neural predictors, we select the simplest, the *bimodal* predictor. This last predictor uses 2 bits to store the history of the branch, and has been proved to be as efficient as using a similar N-bit predictor [36, Chapter 3].

To reduce the penalty for a branch, we need to know the address to fetch the next instruction from by the end of the fetch of the branch instruction itself. Therefore, we need to know if the as of yet undecoded instruction is a branch, and if so, what the new program counter will be. To solve this problem, a *branch target buffer* (BTB) can be maintained. This buffer has a structure very similar to that of a cache, and stores addresses along with whether the instruction at that address is known to be a branch or not. If it is a branch, it also stores the branch prediction (in our case, the bimodal prediction value), and the address of the next instruction. A larger BTB table will lead to near perfect branch prediction, i.e., maximum performance, at the same time occupying a large chip area.

Due to the above considerations, and in keeping with the study of Strydis and Gaydadjiev [77], the Genetic Algorithm can select the branch predictor to be *taken*, *not-taken*, or *bimodal*. If the predictor is selected to be bimodal, the algorithm also selects a BTB size. The latter is done by encoding the *number of sets* and the *associativity* of the BTB table.

The range of the bimodal predictor was selected to be from 32 to 128 entries. This is same as the range used by Strydis[77]; where the 4K upper limit models an ideal (infinite) predictor.

Finally, the Genetic Algorithm also evolves optimum values for the *return address stack* (RAS). The RAS stores the address where a return instruction will point to at the end of a procedure call, by keeping the last address from where a procedure was called at the top of the stack. Since the procedure is likely to return to where it was called from, RASs are highly accurate. We support RAS sizes from 0 to 8.

3.1.3 Caches

A CPU cache is a small and fast memory used for storing copies of instructions and data that are most frequently used for processing. Caches are used to reduce the average time taken to access memory, thereby increasing performance.

Data is stored in caches in contiguous units called *blocks*. In the most general cache implementation, each block can be placed in a restricted set of places in the cache, usually decided by $(Blockaddress)MOD(numberofsets)$; where a *set* is a group of blocks of a fixed size (say n). A block is first mapped onto a set, and then it can be placed anywhere in the set. The size of the sets, n, is known as the associativity [36, Chapter 5], and the cache is called an *n-way* associative cache. Two extreme cases of such cache organization are *direct mapped* caches and *fully associative* caches. In the former, a block can be placed at exactly one location (1 way associative); while the latter allows the block to be stored anywhere in the cache.

As we increase cache sizes, we should observe an increase in performance, up to the point where all the data needed can be found in the cache. After this, no more performance gain can be achieved with bigger cache sizes. Also, as the cache size increases, the access times looking for data in the cache also increase. Therefore, larger caches than required will have a performance penalty. In addition, larger cache sizes consume more power and take up more area. This trade-off behaviour is highly dependent both on the application on hand, and on the input data to the application. This makes it an interesting variable for our study.

In an implant scenario, we need ultra low power and small size. In many similar embedded applications where moderate performance is acceptable, caches might be considered an overkill. However, implant applications are periodic [72], and caches might help in bringing down total energy consumption if they can increase the sleep-wake ratio of the processor by speeding up execution, and thereby reducing the time the processor needs to be awake. In the future, a simulator that supports modeling sleep-wake states could be used for estimating energy, which can be included as one of the objectives.

Note that the simulator we use implements a 32-bit wide architecture. As stated by Strydis [72], an implant processor is more likely to use a smaller word size, proposed in the same work to be approximately 8-bit wide. The author of that work suggests scaling down the 32-bit simulator cache results by 8x to arrive at the corresponding hypothetical 8-bit ISA used by an implant processor. Nevertheless, all cache sizes (and the corresponding objective metrics) in this work are reported as the original 32-bit architecture results.

We use both the Instruction and the Data caches in our optimization process, and represent them each using *block size*, *number of sets*, and *associativity*. The ranges for these are - 1 to 8192 sets, 8 to 32 words block-size, 1 to 32 way associative; where a set size of 1 is a fully associative cache and an associativity of 1 refers to a direct mapped cache. This leads to cache sizes from 8KB to 16MB. Although the simulators theoretically supports a larger range of cache sizes, in practice, it was found that it behaved erratically for larger cache sizes. Therefore, we selected the maximum possible ranges for the three parameters based on this consideration. In addition, bigger cache sizes are unrealistic, and are quite unlikely to found in an embedded application.

We also include the cache-hit latency as a variable. This latency depends on the degree of pipelining in the cache access. We expect to see an increasing cache hit time but a greater penalty on mispredicted branches and more clock cycles between the issue of load and use of the data. Although the simulator theoretically supports variations in both I-cache and D-cache latencies, we found that varying the I-cache latency above the default of 1 often had the simulator hanging. Therefore, we only include D-cache latency as a variable, which we vary from 1 to 16 clock cycles.

Finally, we also use the cache replacement policy as a variable, with the choices being *Least Recently Used* (LRU), *FIFO*, and *Random*. Which policy works with a particular cache is highly dependent on the kind of data in the cache, so we hope to find the policy suitable for our benchmarks for a particular cache configuration.

Note that the above discussion pertains to L1 caches. At this time, low power embedded processors hardly have even an L1 cache; hence in our opinion, L2 caches would be an overkill, and therefore exclude them from our exploration. Other features of the simulator that we switch off are cache flushing and I-cache compression, since both these techniques also require more hardware complexity.

3.1.4 Buffers

Write and Fill buffers

The XTREM simulator we use models the Intel Xscale core, which includes two buffers, the write buffer and the fill buffer. These buffers act as intermediaries between the processor's core and the main memory, and help achieve better performance by decreasing the stalls due to accessing the main memory [11]. This is especially important because of the high clock rates used by XTREM. The final implant application is likely to have a much slower clock rate and lesser demands on performance. Therefore, we decided to keep the buffer sizes fixed at the minimum supported by XTREM, i.e., exactly two entries.

Parameter		Encoded		Decoding	
Name (Ref)	Actual Range	Range	Bits	Formula	Remarks
Core Clock Frequency ($Freq$)	[1...64]	[0...63]	6	$n + 1$	Not used in current version
Branch Prediction ($Bpred$)	$Bimodal, Taken, notTaken$	[0...2]	2	-	$bit_0 = isBimod$, $bit_1 = isTaken$
Branch Target Buffer: Number of Sets (btb_{nsets})	[32...128]	[0...5]	3	2^{n+5}	Only valid for $isBimod = TRUE$
Branch Target Buffer: Associativity (btb_{assoc})	[1...32]	[0...5]	3	2^n	Only valid for $isBimod = TRUE$
Branch Prediction: Return Address Stack (RAS)	[0...8]	[0...4]	3	$floor(2^{n-1})$	-
L1 D-Cache: Number of Sets (D_{nsets})	[1...8192]	[0...13]	4	2^n	-
L1 D-Cache: Block Size (D_{bsize})	[8...32]	[0...2]	2	2^{n+3}	-
L1 D-Cache: Associativity (D_{assoc})	[1...32]	[0...2]	3	2^n	-
L1 D-Cache: Replacement Policy (D_{repl})	f, r, l	[0...2]	2	-	$bit_0 = isFifo$, $bit_1 = isRandom$
L1 D-Cache: Latency ($D_{latency}$)	[1...16]	[0...4]	3	$floor(2^n)$	-
L1 I-Cache: Number of Sets (I_{nsets})	[1...8192]	[0...13]	4	2^n	-
L1 I-Cache: Block Size (I_{bsize})	[8...32]	[0...2]	2	2^{n+3}	-
L1 I-Cache: Associativity (I_{assoc})	[1...32]	[0...2]	3	2^n	-
L1 I-Cache: Replacement Policy (I_{repl})	f, r, l	[0...2]	2	-	$bit_0 = isFifo$, $bit_1 = isRandom$
L1 I-Cache: Latency ($I_{latency}$)	[1...16]	[0...4]	3	$floor(2^n)$	Not used in current version

Table 3.1: Processor Design Parameters considered in this work

Translation Lookaside Buffer

Translation lookaside buffers (TLB) are special address translation caches [36] to speed up main memory accesses by caching page translations. We keep the smallest allowed values for the Instruction and Data TLB buffers, with exactly a single entry for each.

3.1.5 ALUs

The ImpBenchmark suite uses only a moderate number of arithmetic operations [78, Figure 4a]. Out of these operations, the number of multiplication operations is likely to be even lower. Keeping this in mind, we fixed the number of ALUs and Multipliers in our design to a single unit of each kind. In future, simulator support can be added to omit even the multiplier in case a more minimalist design is required.

An approximation we make when using the XTREM simulator is to include exactly one *Floating Point Unit* (FPU) and one *Floating Point Multiplier* (FPM) in the design. This is because the simulator we use only supports one or more FPUs and FPMs, not zero. However, an implant is unlikely to have these units altogether. Therefore, we need to minimize the impact of these extra units by avoiding floating point calculations. None of the instructions executed by the simulator running the ImpBench benchmarks were observed to be floating point instructions, thereby justifying this approximation. Note that since we approximate the processor area by the cache sizes, these additional units have no impact on the area.

3.1.6 Other Options Considered

Memory Ports is another feature of the XTREM simulator that cannot be disabled altogether. Therefore, we selected the minimum ports, namely two. We decided to also omit Memory Pipelining from the design. As with other features we have pared down or disabled, we believe these two features go against the requirements of our processor.

3.2 Chromosome Encoding

In the previous subsection, we listed the design parameters we are interested in for performing the design space exploration. Genetic Algorithms optimize the information encoded in the chromosomes of the individuals. Therefore, we need to define an encoding for the processor parameters to convert them into a chromosomal representation that the Genetic Algorithm can work with.

There are several chromosomal encoding strategies. The simplest is encoding each variable as a string of 1 and 0 bits. Even in this representation, we can choose to either directly represent the bits; or use a Gray-coded representation if we need to avoid large jumps in the variable values and the mutation operation is found to be too disruptive for a particular problem.

In our implementation, the mutation value is set such that only one bit is expected to change per chromosome between successive generations. If we use a Gray coded representation, this would mean that mutation would lead to a set of variable that is only distance 1 from the previous set of variables - for example, all other variables being equal, the cache size of the child is the exactly next lower size allowed than the cache size of the parent. Since we are interested in obtaining a good Pareto-spread, rather than in perfecting the global optimum found, Gray coding will take too long to evolve diverse results. Also, we use an elitist strategy that saves previous good results, so even if the mutation and crossover operations happen to be a little disruptive, good solutions will still get preserved. Therefore, we use the simple encoded approach. Table 3.1 lists the details of our chromosome encoding - the variables chosen, their ranges, and the encoding and decoding rules.

An interesting experiment for future work would be to encode the chromosome with real numbers rather than a binary bit string, and thereby ensure that the parameters get perturbed independent of each other during crossover. This might affect the convergence rate – for better or for worse is highly dependant on the design space.

3.3 Objectives

Gries [31], in his extensive study of processor design, lists the possible design goals that can be used for optimization of processors. He lists the primary objectives as cost, power dissipation, speed, and flexibility; with the last being a meta-quality that arises from considerations such as programmability or reconfigurability. Except for this last quality, we model the other three design goals as our *objectives* – with the cost metric represented as area, power dissipation as average power, and speed as IPC. For future work, energy consumption or combined metrics such as the energy-delay product can be also be explicitly included as design goals.

Each candidate solution for finding the Pareto front is represented as an individual. The variables forming the candidate solution are represented as the chromosome of the corresponding individual. To judge the "goodness" of the candidate solution, we need to evaluate this solution with respect to the objectives of the problem - power, performance, and area.

As we mentioned in the previously, we use the XTREM and CACTI simulators for evaluating the objectives. We use a modified version of the XTREM simulator such that it gives the breakdown for (average) power consumed by different functional units every 10,000 cycles. For this work, we consider the sum of these power values (averaged across the benchmarks for multiple benchmarks) as the *power* objective. We represent the *performance* objective as *Instructions per Cycle* (IPC), which is also returned by XTREM. The CACTI simulator is used to find the area occupied by different configurations of the included cache structures. We also use the simulator to approximate the area occupied by the branch predictor - this is justified, as branch prediction tables have similar implementation structure as caches. We use the sum of the I-cache, D-cache and branch predictor area values as an approximation of the area of the entire processor. The approximation error is the area of the other components of the processor, for example, the ALU. However, the absolute error is quite small because the caches form a sizeable component of the total processor area. In addition, this error is nearly constant for different configurations that we examine because we only vary the caches and branch prediction schemes, the rest of the processor components remain the same. Since the error remains nearly constant, it minimally affects the correctness of the Pareto points found because the algorithm we use requires a *relative* measure of the fitness objectives rather than *absolute* values.

The flavour of Genetic Algorithm that we use is a minimizing algorithm. However, our objectives are minimizing area and power while maximizing performance. To convert our problem to a fully minimizing problem, we need to take the complement of performance as the objective encoded in our algorithm. We use IPC as the metric of performance, which is an integer number. Therefore, we can simply use IPC*(-1) as the objective to be minimized. Note that cycles per instruction (CPI) could have also been used: since CPI is the inverse of IPC, it would give an identical relative ordering of processor configurations as IPC*(-1).

3.4 Constraints

Most of the problem constraints (ranges, allowed values - such as powers of 2) have been implicitly handled in the chromosome encoding. The missing constraint is that the number of sets must always be less than the associativity. Therefore, this is modeled as an explicit constraint for Bimodal table, Data cache and Instruction cache.

3.5 Benchmarks & Data Sets

We already gave an overview of the benchmarks and data sets used in this work in Chapter 2, Section 2.3.5. In this section we describe each benchmark in more detail.

For our main analysis (Chapter 4,Section 4.3), we optimize the algorithm for the 8 original ImpBench benchmarks (i.e. excluding the 'stress' benchmarks):

MiniLZO [55] (abbrev. *"mlzo"*) is a light-weight subset of the LZO library (LZ77-variant). LZO is a data compression library suitable for data de-/compression in real-time, i.e. it favors speed over compression ratio. LZO is written in ANSI C and is designed to be portable across platforms. MiniLZO implements the LZO1X-1 compressor and both the standard and safe LZO1X decompressor.

Finnish [17] (abbrev. *"fin"*) is a C version of the Finnish submission to the Dr. Dobb's compression contest. It is considered to be one of the fastest DOS compressors and is, in fact, a LZ77-variant, its functionality based on a 2-character memory window.

MISTY1 (abbrev. *"misty"*) is one of the CRYPTREC-recommended 64-bit ciphers and is the predecessor of KASUMI, the 3GPP-endorsed encryption algorithm [48]. It is designed for high-speed implementations on hardware as well as software platforms by using only logical operations and table lookups. Also, MISTY1 is a royalty-free open standard documented in RFC2994 [56] and is considered secure with full 8 rounds.

RC6 [48] (abbrev. *"rc6"*) is a parameterized cipher and has a small code size. RC6 is one of the five finalists that competed in the AES challenge and has reasonable performance. RC6-32/20/16 with 20 rounds is considered secure.

Checksum [7] (abbrev. *"checksum"*) is an error-detecting code that is mainly used in network protocols (e.g. IP and TCP header checksum). The checksum is calculated by adding the bytes of the data, adding the carry bits to the least significant bytes and then getting the two's complement of the results. The main advantage of the checksum code is that it can be easily implemented using an adder. The main disadvantage is that it cannot detect some types of errors (e.g. reordering the data bytes). In the proposed benchmark, a 16-bit checksum code has been selected which is the most common type used for telecommunications protocols.

CRC32 [1] (abbrev. *"crc32"*) is the Cyclic-Redundancy Check (CRC) is an error-detecting code that is based on polynomial division. In the proposed benchmark, the 32-degree polynomial[1] specified in the Ethernet and ATM Adaptation Layer 5 (AAL-5) protocol standards has been selected.

Motion (abbrev. *"motion"*) is a synthetic benchmark based on the algorithm described in the work of Wouters et al. [87]. It is a motion-detection algorithm for the movement of animals. In this algorithm, the degree of activity is actually monitored rather than the exact value of the amplitude of the activity signal. That is, the percentage of samples above a set threshold value in a given monitoring window. In effect, this motion-detection algorithm is a smart, efficient, data-reduction algorithm.

DMU4 (abbrev. *"dmu4"*), is a synthetic benchmark based on the system described in the work of Cross et al. [13]. It simulates a drug-delivery & monitoring unit (DMU). This program does not and cannot simulate all real-time time aspects of the actual (interrupt-driven) system, such as sensor/actuator-specific control, low-level functionality, transceiver operation and so on. Nonetheless, the emphasis here is on the operations performed by the implant core in response to external and internal events (i.e. inter-

[1]CRC32 generator polynomial: $x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^{8}+x^{7}+x^{5}+x^{4}+x^{2}+x+1$.

rupts). A realistic model has been built imitating the real system as closely as possible.

DMU3 (abbrev. *"dmu3"*) is an extension of *"dmu4"* [79]. The original *"dmu4"* benchmark emulates a real-time implantable system by reading real pressure, temperature and integrated-current sensory data (as provided by true field measurements by the implant developers [13]) and by writing to transceiver module (abstracted as a file). *"dmu3"* emulates this in a more sophisticated manner by also accurately modeling the gascell unit used to switch drug delivery in the implant on and off. To this end, it reads additional input data termed "gascell override switch" and "gascell override value". The suffix numbers in both *DMU* benchmarks originate from multiple field-test runs using different drug-delivery profiles: A high-low-high varying, "bathtub" profile (#4) has been used for *"dmu4"* and a constant, flat profile (#3) has been used for *"dmu3"*. We introduce it briefly here, so that the content of stressmark *"stressdmu3"* will be better understood.

Stressmotion (abbrev. *"stressmotion"*) and **stressDMU3** (abbrev. *"stressdmu3"*) constitute a new additions to the ImpBench suite (as described in [79]), their creation stemming from the fact that the original *"motion"* and *"dmu4"* benchmarks have considerably long run times w.r.t. the rest of the benchmarks (see Table 4.1). *"dmu3"* rather than *"dmu4"* has been used for extracting a stressmark due to its more sophisticated emulation of the DMU applications. Since all benchmarks essentially are pieces of continuously iterated code, each stressmark is a derived, worst-case iteration of its respective benchmark. That is, an iteration wherein the implant is required to perform all possible operations; thus, the term "stressmark". The stressmarks feature significantly shorter execution times than their original counterparts.

When we consider realtime constraints in Section 4.5, we use the miniLZO, rc6, checksum, and the 2 stress benchmarks to calculate single loop execution times, but the algorithm still optimizes the metrics of the original ImpBench benchmarks.

3.6 Population Size

The Pareto points that form the Pareto front are discrete. Ideally, we would like to have all the Pareto-points possible in the result. However, we do not know how many such points exist in the true front. Also, we may not need to know all possible points, in design, as long as we have sufficient number of points spread over the objective space. The number of points we choose to include is a design tradeoff - the more points we include, the better approximation of the front we will have, but we will need to evaluate more configurations.

The population of a Genetic Algorithm represents the set of candidate solutions. The size of the population represents the maximum number of Pareto points that algorithm can find. If we pick too small a size for the population, we would have lesser tradeoffs available. On the other hand, the Genetic Algorithm is $O(mN^2)$, where m is the number of objectives and N is the population size. So, increasing the population size would have a significant impact on processing time.

Additional limiting factors on choosing the population size was that the algorithm could only accommodate multiples of 4 as the size (to facilitate the selection function, sorting etc.), and that the number of parallel machines available was 11 (For details on Parallelization, please refer to Section 3.11). These machines were also prone to being unavailable, and assuming minimum 5 machines at any given time, we can guarantee that a given generation would finish in $\lceil N/16 \rceil$ to $\lceil N/5 \rceil$ parallel steps. The estimated run-time of the algorithm is directly proportional to the number of parallel steps, is then given by $T_{avg} * P * G$, where, T_{avg} is the average time per parallel step, P is the number of parallel steps per generation and G is the number of generations.

We chose the population size as 20 for our experiments. We found it gave a good spread, while not taking up too much time. In this case, there are $\lceil 20/16 \rceil = 2$ to $\lceil 20/5 \rceil = 4$ parallel steps per generation depending on the number of machines available. In future, the population size can be increased with a hope for a better spread of Pareto points, ideally with a corresponding increase in computation resources.

3.7 Number of Generations

The number of generations in a Genetic Algorithm are the number of iterations the program goes through in searching for the optimal solution. Since they are heuristic algorithms, with an element of randomness to them, Genetic Algorithms can never guarantee that they will find the true global optimum. They can however guarantee that a solution will get *no worse* using strategies such as population saving and elitism. The more iterations we have of the Genetic Algorithm, the likelier it is that a better approximation of the global optimum is found. Therefore, we need to strike a balance between the number of iterations (hence the total computing time), and the quality of the solution found. This can be done by looking at the *convergence* of the algorithm. Although this depends heavily on the type of problem at hand, typically, Genetic Algorithms show a rapid improvement in the solution found in the first few iterations, after which slower improvements are seen; the latter stage sometimes characterized by jumps in the solution when suddenly a better solution is found. The number of generations can be set in a single optimization case as follows - run the algorithm for a limited number of generations or for a reduced problem set, examine the convergence behavior (how fast it converges, how long it stays in a stable state without variation in quality of results), run the algorithm again with the original problem based these observations from the limited problem. Therefore, we ran our experiments on the checksum problem. The detailed experiment setup and findings can be found in Chapter 4, Section 4.1.

3.8 Mutation

Mutation in genetics is defined as "*any alteration in the inherited nucleic acid sequence of the genotype of an organism*" [60]. This alteration being random, can give the new organism either an evolutionary advantage or disadvantage with respect to the parent

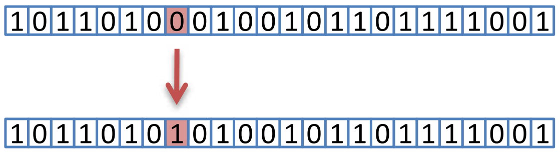


Figure 3.2: Mutation on a sample chromosome – each bit is either toggled or left intact depending on the mutation probability p_m

population and its peers. If the mutation is advantageous, natural selection ensures that the new set of genes get passed on to the subsequent generations. On the other hand, disadvantageous mutations will lead to the organisms possessing the altered gene to die out. Hence, mutation helps create diversity in the gene pool and eventually leads to better organisms.

From the above, we see that mutation would be useful also in our adaptation of the Evolutionary process. The goal of the genetic algorithm is to find better solutions to a problem. Mutation leads the algorithm into new and unexplored directions, thereby preventing the search from stagnating at local optima [30]. There are several strategies to implement mutation in genetic algorithms. All of them involve randomly changing the chromosomes of the parent population in some way when creating the child population from these chromosomes. Mutation in our binary encoded chromosome can simply be defined as random "bit flips" in the encoded value (Figure 3.2). The simplest way to implement this is to select a mutation probability p_m for the algorithm. At the mutation stage, a random number (r_b) between 0 and 1 can then be generated for every bit (b) in the chromosome. If r_b is less than p_m, then bit b is toggled. Obviously, the higher the value of the mutation probability, the more the number of bits in the chromosome that are likely to be flipped.

Although mutation is beneficial, too much change in the chromosomal structure disrupts the evolutionary process by destroying the good information evolved and stored in the previous populations' chromosomes. In fact, a very high rate of mutation degenerates the algorithm into a very complicated random search algorithm. Therefore, it is important to select a good value for the mutation probability p_m. Note that even though for every problem there might be an optimal value of p_m for which the algorithm is likely to converge the fastest; the class of Evolutionary Algorithms by nature are quite robust to parametrical changes and even a sufficiently good value of p_m will lead to good convergence rates. In such cases, the quality of solution found will not be affected, though the algorithm might take more time to arrive at the solution. This is property is important because Genetic Algorithms are random by nature - different runs of the same algorithm might produce different results. Therefore, optimal convergence rates are hard to find, and consequently, optimal mutation probabilities: we need to make do with "good enough" value of p_m.

There are a number of ways to select the mutation probability, including complicated adaptive mutation strategies [42]. However, the simplest and most recommended strategy is simply setting the mutation probability as $p_m = 1/n$ where n is the chromosome length [5] [54]. This means that a single bit per chromosome is expected to change from the parent to the child population. Therefore, the child chromosome is likely to vary from the parent chromosome in exactly one attribute, that too by a hamming distance of one. We use this approach for setting the mutation probability in our implementation of the Genetic Algorithm.

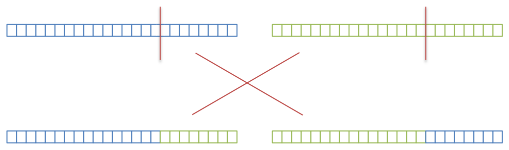
3.9 Crossover

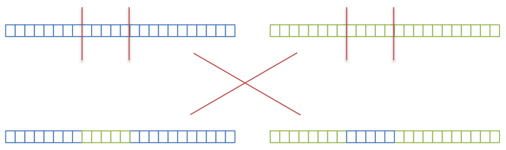
In nature, the parents's genes combine to form the child's genes, and the child inherits some characteristics of each parent. This is an important evolutionary process - the genes evolved up to the parent population are passed on to the child population in a recycled manner. Consider desirable gene variations A and B, and undesirable variations A' and B'. Some children may end up inheriting gene A from one parent and gene B from another, leading to an overall *better* individual, on the other hand, some children will lead to an evolutionarily disadvantaged individual with genes A' and B'. Eventually, the latter will get weeded out from the population while the former will get to reproduce more and pass on this good combination of genes, hence leading to an overall increase in the quality of individuals.

This idea is important for Genetic Algorithms. Our goal is to solve a search problem spread over several input values. In many problems, some particular input combinations might be partial solutions to the problem [39]. By randomly combining part of the input variables (encoded as a section of the chromosome) from one candidate solution (parent X) to a part of the input variables from another candidate solution (parent Y), it may be likely that partial solutions from both chromosomes get included in a new chromosome that solves the problem better than both X and Y.

Used carefully, crossover can lead to much quicker evolution times, and is central to the idea of Genetic Algorithms examining more solutions in the more promising regions of the solution space [39]. Several kinds of crossover techniques have been proposed, with one book listing over 50 different techniques [32]. However, the main techniques considered in literature are restricted to single-point [39], 2-point crossover, multi-point, variable-to-variable, uniform [80] and adaptive crossover [69]. Figure 3.3 illustrates some of these techniques. However, these techniques and studies comparing these techniques such as by Hasançebi and Erbatur [34] focus on single-objective optimization. Although studies such as [41] have been done on techniques for real-coded chromosomes in multi-objective optimization, we could not find similar studies on binary coded chromosomes. Therefore, we use the single-optimization studies as a guideline towards choosing the crossover strategy.

However, the results from the studies and strategies mentioned above do not provide any clear consensus on which of the crossover strategies is optimal. This may be because crossover strategies seem to be affected by the problem domain. However, 2-point crossover seems to perform reasonably well across problem domains. In addition, it is a


(a) If a certain crossover probability is met, a single crossover point is randomly chosen. All data after this point is exchanged between the two parent chromosomes


(b) If a certain crossover probability is met, two random crossover points are selected. All data between these points is exchanged amongst the parent chromosomes

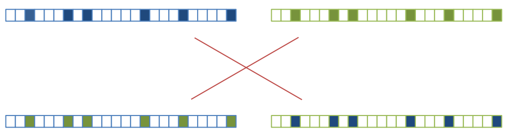

(c) Each gene of the parents is exchanged with a certain crossover probability

Figure 3.3: Different crossover techniques, with each square representing a single gene (bit) in the chromosome. The top two chromosomes in each figure are the parent chromosomes, and the bottom two are the resulting children after the crossover operation is applied

simple but powerful technique. Therefore, we use 2-point crossover. Note that although this is possibly not the optimal strategy for our particular problem, Genetic Algorithms are robust enough to perform well even with good but non-optimal configurations.

In addition to the crossover strategy, it is also important to set a good ”rate of crossover”. This rate determines the percentage of chromosomes that undergo recombination.

If the rate is too high, we may end up destroying too much information. On the other hand, a high rate is likely to lead to more diversity in the solutions found. Unlike mutation, crossover rates cannot be chosen simply from the chromosome size. They are much more domain specific. We perform experiments to select crossover probability in Chapter 4.

3.10 Selection

Natural selection is the foundation of evolution. In nature, genes of different organisms compete to bring their respective organisms an evolutionary advantage over others [39]. For example genes that allow an animal to sense its predator from a larger distance than others of its species may allow the animal to survive longer and have more offspring, who in turn inherit this trait. Over generations, it is likely that the less-adapted animals die off and there exist more animals with the particular advantageous gene. Hence, the population as a whole improves.

The rate at which the advantageous genes described above favored over other variations is termed *selective pressure*. The degree of selective pressure depends on environmental factors for example the availability of natural resources. If the competition to survive is higher, such as when there is a shortage of food, the selective pressure is higher and better adapted organisms are much more favored. In times of plenty, even less competitive organisms may survive. This process is critical to evolution. If the selective pressure is too high, then evolution becomes extremely directed towards organisms solely adapted to that particular pressure, and gene diversity is sacrificed. If the pressure is too low, meaningful evolution cannot take place, since no organism is favored over the other.

In terms of a genetic algorithm, selective pressure determines the direction the search takes. A very high bias towards favorable individuals may lead to stagnation at local optima points due to loss of search diversity. On the other hand, no bias reduces the problem to a blind random search, and prevents the algorithm from converging.

The selection process is central to the NSGA-II algorithm, and distinguishes it from other multi-objective genetic algorithms. The authors of the algorithm perform selection by sorting the population using a *Non-dominated sorting* algorithm, and then using a binary tournament selector to decide which individuals are to reproduce. We left this process essentially unchanged from how the authors proposed it.

3.11 Parallelization & Optimization

We found that evaluating a single individual with respect to the 8 ImpBench benchmarks and all 7 workload sizes and two input data sizes of $10KB$ and $1KB$, took about 900 seconds on average. Therefore, given a population size of 20 and 200 generations, a single run of the proposed algorithm would take $20 * 200 * 900/(3600 * 24) \approx 42$ days to finish. This runtime is quite unacceptable and needs considerable speeding up in order to be feasible. It is this reason that we reduced the workload sizes to only consider the worst case workload of 10KB EMGII. This reduced a single evaluation to ~ 126 seconds, thereby taking down the time taken to finish to about 6 days. However, this is also quite long and needs more speeding up. One of the most promising techniques to make GAs faster is parallelization.

There are several different techniques proposed in literature to parallelize genetic algorithms [29], which broadly fall into three major categories. Using terminology of Cantú-Paz [8], these are (1) Global single-population master-slave GAs, (2) Single-population fine-grained GAs, and (3) multiple-population coarse-grained GAs. The master slave approach is the functional equivalent of the non-parallel GAs. The master performs the selection, mutation and crossover. Only the evaluation of the fitness function is parallelized amongst the slaves. Since the entire population is considered in selection, the GA is called "Global". In the second parallelization approach, each processor node is (ideally) assigned exactly one individual and is responsible for evaluation, and mutation of that individual. Crossover can only occur between nodes that are close together. This process differs from the non-parallel version in that the global population is never taken into account, there are just local interactions. Finally, in the multi-population approach, also known as the island model, several sub-populations are maintained, (ideally) one population per node. Each of these sub-populations evolve more or less independently of each other with only occasional individual exchanges known as *migration*.

Out of these three approaches, the first approach is the simplest to implement and does not change the character of the Genetic Algorithm. Moreover, it is more suitable for our problem given that the fitness evaluation is more time consuming by orders of magnitude than other evolutionary steps, the dynamic nature of resources available (as we shall see) and the small population size we chose. Therefore, we use the master-slave approach to parallelize our Genetic Algorithm.

Ideally, in the master-slave parallelization, one would require as many computation nodes available as the population size, i.e. 20 nodes. However, we did not have as many processors available. Therefore, the final parallelization was as follows:

$$\begin{array}{llll}
\text{Let} & : \ \textit{Population size} & = N & \\
\text{Also, let} & : \ \textit{Number of processors} & = P & \\
 & & = \textit{Number of configurations evaluted per step} & \\
 & \Rightarrow \ \textit{Number of time steps} & = N/P & \{\text{Where } P \text{ is a factor of } N\} \\
\text{Note that} & : \ \textit{Ideal speedup} & = N & \\
 & \therefore \ \textit{Actual Speedup} & \approx N/(N/P) = P & \{\text{Where } P \text{ is a factor of } N\}
\end{array}$$

3.11.1 Parallelization Software

Several software libraries have been developed for writing parallel programs. the most popular libraries seem to be MPI (Message Passing Library), RPC (Remote Procedure Call), PVM (Parallel Virtual Machine) and Java RMI (Remote Method Invocation) [50] [62]. Out of these, RPC forms the base of most modern distributed software packages, and is an integral part of all Unix based operating systems and is used in many system services such as the Network File System (NFS) and the Network Information Service (NIS). It has also been implemented for other platforms. Although RPC does not have advanced features of the other libraries such as Load Balancing (MPI, PVM), topology support (MPI) etc, it has a lower latency and better bandwidth utilization than the other libraries. It turns out that we do not need the advanced features and the simple client-server paradigm of RPC suffices for our needs. Moreover, our objective was to parallelize the Genetic Algorithm written in C++ without much change to the base structure and using RPC and the associated tools simply meant the fitness function could now be called as a remote procedure on multiple machines instead of a local call.

Therefore, we use RPC for the parallelization of our GA code.

It should be mentioned that more complicated middleware such as CORBA also exist for parallelization, but in our simple procedural program, these seem to be an overkill and therefore were not selected for this work. However, it must be noted that a framework such as Condor (a high-throughput computing software framework for coarse-grained distributed parallelization of computationally intensive tasks)[82], which can transparently scavenge idle desktop computation cycles from a given heterogeneous network, could be an ideal software for future implementations of this project.

3.11.2 RPC

RPC provides an interface that allows the programmer to call procedures on remote machines as if they were local procedure calls. The caller is known as the *Client* and the callee is known as the *Server* Figure 3.4. The programmer need only implement the client program and the procedure at the server end. The rest of the code needed for the actual data transfer can be created by using a protocol compiler known as RPCGEN[52]. RPCGEN automatically generates remote program interface modules based on descriptions written in the RPC programming language, which resembles C in syntax and semantics, and the output code is also generated in C language. In our case, the RPC description file consists of a description of the chromosomal data, the additional variables used to signal various error conditions and the program and version numbers. The latter numbers uniquely identify the kind of server process and reside in the RPC registry of the server. Potential clients can query this registry to find a list of appropriate procedures that can be called. The program has been written to take full advantage of this fact. Automatic scripts have been written that periodically check the availability of free server machines by checking the RPC registry of candidate machines with an option to start the server processes remotely on available machines if not already started. It is this list that is read periodically by the client to determine available machines.

RPC calls are synchronous (blocking), with the maximum call duration is bound by a timeout value. However, multiple calls can still be made using forking/threads. In our program, we use forking on the client side to spawn several processes, equal to the number of free servers available at that moment. Each of these child processes makes a procedure call to the *fitness function* implemented on a single server and blocks for the duration of the call. The result from the server is then communicated via software piping back to the main process. Note that we refer to 'clients' and 'servers' according to the RPC naming conventions, and therefore our design consists of a single client and several servers. This is a little counter-intuitive to the traditional notion of multiple clients serviced by a single server.

Besides ease of development, one of the advantages of using RPCGEN is also that the client and servers interact through the standard interfaces created by RPCGEN. This means that a particular client can work with multiple implementations of the server on different platforms. This makes the design very modular and flexible - If for

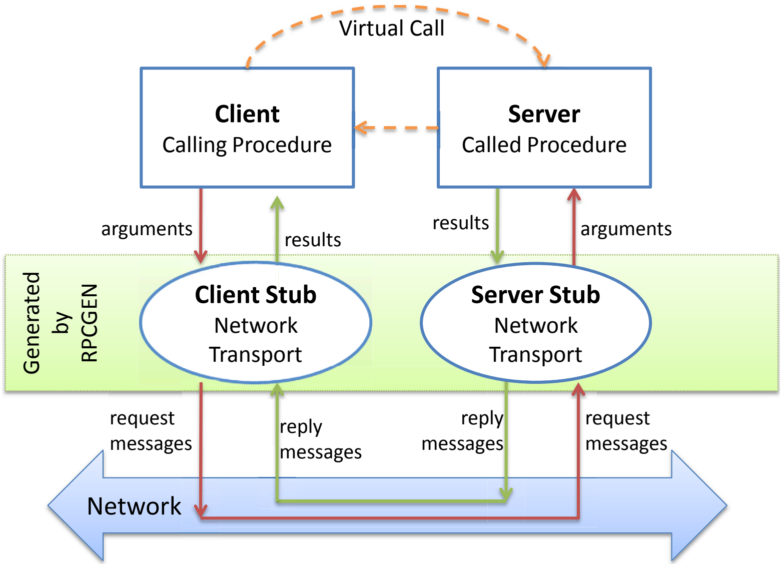


Figure 3.4: Schematic of a remote procedure call

example the processor configurations need to be evaluated differently, or using different simulators, then only the server implementations need to be changed; the client may only need an adjustment in the timeout values.

In order to identify the overheads of parallelization, we did some timing measurements[2] of different parts of the code. Successive calls to the timer without any code in between gave an average value of $1e-06$ seconds. This is the error of the timer, and must be kept in mind when comparing actual measurements. An empty RPC call [3], where a client calls a server function, which returns immediately, gave an average of 0.00036 seconds. This time includes the time for establishing connection, marshalling data, data transfer, and coping standard values to the return object at the server. Finally, the one-time average IPC setup time (starting and registry of server process) was 0.130 seconds and the parallelization overhead on the client (housekeeping of free machines available, forking etc.) was 0.130 seconds. Clearly, all these overheads are negligible compared to the execution time of the simulator.

3.11.3 Computing Resources Available

Due to administrative reasons, the resources available for computation were limited to the laboratory workstations. These workstations are networked and shared by several users. Therefore, we needed to setup the system to utilize the idle computation resources of the workstations without disturbing other users. Therefore, the experiments were run as low priority processes. Furthermore, these computers being public, could be switched on or off anytime. To guard against this, the client process was run on a machine that

[2]all measurements are averages with a sample size of 20
[3]both on localhost and on other machines

was not allowed to switch off, and, a script was run on this "always-on" machine that periodically checked for machines that were on, but did not have the server process running. The script automatically started the server process on these machines which would then be discoverable by another script that periodically fed a list of available machines to the client. In addition to the need for switching on server process to increase availability, it might also happen that the server process dies during a call from the client, for example due to the machine being switched off. If the server exits gracefully, the client gets back an error code; if it does not, the client will experience a timeout with respect to that particular server. To handle this, the client has been configured to re-execute failed server calls. However, to guard against too many re-executions due to errors such as the simulator failing at some input combination, only a single re-execution per individual is allowed. If even the re-execution attempt fails, that individual is marked as having a constraint violation and hence gets weeded out of the population by the algorithm. The estimation for total execution time should include the (optional) re-execution time for each generation.

An unexpected obstacle we faced was that the initial machines used for testing ran Fedora 8 Linux. On these machines, the RPC services could be started and listed by all users. However, on machines running newer versions of Linux, root privileges were needed. Due to security reasons, these could not be enabled. Therefore, the configurations finally used were AMD Athlon(TM) XP 2400+ @ 2000.244 MHz, with a cache size of 256 KB, running Fedora 8 Linux. Note that one of the factors in the large computation time is certainly the age of these machines.

3.12 Real Time Constraints

Real life implants are likely to have real-time demands on the processor performance. For example, the processor must gather sensor data periodically, process it and perform other tasks such as data transmission and storage. All this, it must do before the next batch of data becomes available. In such cases it is important to design the processor with these constraints in mind. In terms of design points, this means that some lower-performing processors, would be eliminated from the trade-off spectrum even though they are might have good power characteristics. In order to demonstrate how real-time constraints can be incorporated into our design, we chose an artificial constraint of 1 minute. As we shall see, this was a very loose constraint and did not shape the results appreciably. Therefore, in order to further study the impact of realtime constraints, we then ran the experiment for several other constraint values.

A conceptual implant loop given by Strydis[74] is shown in Figure 3.5. We chose *miniLZO*, *rc6* and *checksum* as the compression, encrypion and data-integrity applications respectively. The 'DMU application' in the block diagram actually refers to additional processing applications. Keeping this in mind, we substitue DMU in the block diagram with the newly created *stressbenchmarks*. As described previously, the stress benchmarks constitute the worst case single iteration of the *DMU* and *motion* benchmarks, processing 10KB of data. For each processor configuration, we run these 5

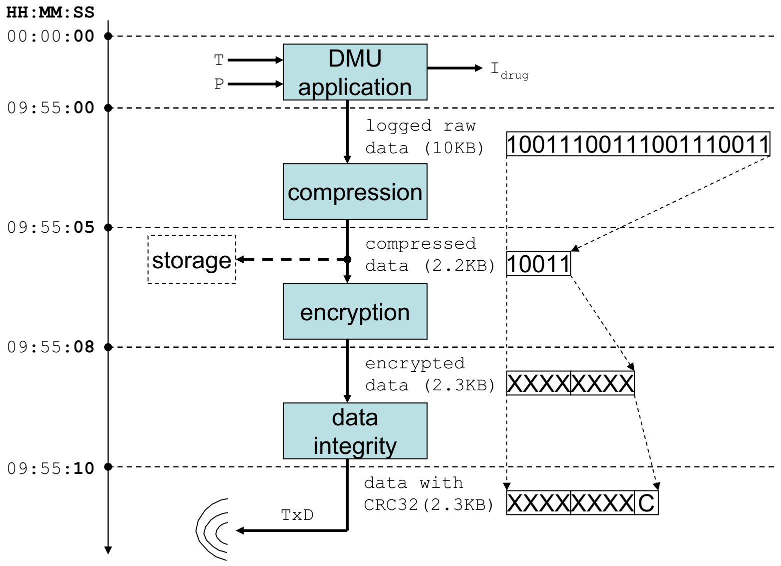


Figure 3.5: Conceptual execution cycle of a typical implant application

applications and find the total simulated time (i.e. time it would take for the benchmarks to run on a real processor with that particular configuration). If this time violates the realtime deadline, a constraint violation is generated, and that configuration is deemed unacceptable. However, the metric the algorithm optimizes remains the original 8 benchmarks of the ImpBench suite – so that comparisons can be made between the two results.

3.13 Additional Features

Even though automated scripts take care of managing the client and servers, failures do occur occasionally. In such a case, it is helpful to be notified immediately of the error than manually checking on all the clients and servers, specially considering the long execution time of the program. Therefore, the client and server have a configurable feature to notify by email such errors with an error report and detailed log files. It is also useful to keep track of the results, and these and the associated plot files can also be emailed every few generations.

As mentioned before, processor evaluation takes up a significant time. Therefore, the problem might benefit from caching the objectives of previously evaluated individuals from past generations in case the individuals survived in future generations. Therefore, no individual would be evaluated twice. Since this was done with the inbuilt C++ 'map' class, the cache lookup was very efficient and justified the extra time it took every population evaluation. This version with result caching was run 5 times to take into account the fact that different runs evolve different individuals. It was observed that in practice, the actual number of cache hits was only about 2% of the individuals evaluated. Therefore, caches do not benefit this particular problem, and were turned off for subsequent runs. This result also shows that in nearly all population transitions, the newer individuals created through mutation and crossover were found to be better by the selection mechanism.

Experimental Results 4

Before we discuss the results, it may be helpful to visualize the sort of results that are expected for the problem. Consider a hypothetical problem that has as its Pareto-minimum front a plane (say $\mathbb{P}$) cutting across the x,y and z axes as shown in figure 4.1. Let this Pareto front be unknown, and that we wish to approximate this front through a genetic algorithm setup as described in this work. By definition, since $\mathbb{P}$ is the Pareto-minimum front, all other points that satisfy the problem must be greater than or equal to the points on $\mathbb{P}$. Therefore, only the region above and including $\mathbb{P}$ is the feasible region. The genetic algorithm starts with randomly chosen (feasible) points in the solution domain which forms an approximation (say $\mathbb{P}'$) of the front we are seeking. In subsequent generations, the algorithm tries to modify points included in $\mathbb{P}'$ such that $\mathbb{P}'$ moves closer and closer to $\mathbb{P}$. After a sufficient number of iterations, we expect the algorithm to have moved close enough to the true front for our purposes, and we finish this iterative process.

Note that although our problem may not have a straight forward front as in the above mentioned hypothetical problem, we should be able to observe the solution fronts converging in a particular direction, towards a (not necessarily regular) surface.

The population size of the genetic algorithm restricts the number of points $\mathbb{N}'$ in $\mathbb{P}'$. In theory, $\mathbb{P}$ has $\mathbb{N}|\mathbb{N} >> \mathbb{N}'$ points. Therefore, in order to be a good solution, $\mathbb{P}'$ must not only be close to $\mathbb{P}$, it must also capture the shape of $\mathbb{P}$ accurately. The latter is ensured if the points in $\mathbb{P}'$ are spread out evenly over a large range of values. These two qualities are independent of each other, and we must take into consideration both of them when comparing potential solutions.

Figure 4.2 shows a more realistic design space and Pareto-front for a minimization problem. Note that the Pareto-optimal front is discontinuous. Figures 4.3a and 4.3b each show two potential solutions, the former with design points that are already at the actual front, but they are clustered and the diversity is not so good. On the other hand, the latter solution has a good diversity but the convergence is not that good. A better solution would be one that strikes a good balance between these two extremes.

So far, we have only talked qualitatively about the expected solutions. However, we also need a way to quantitatively describe the relative merits of the solutions obtained. Most of the existing methods use the actual Pareto points as a reference to evaluate the merit of a solution. However, in a real-world optimization problem like ours, the true Pareto front is not known. In this case, we can not directly compare the solutions to the optimal Pareto-set to gauge their relative merits. Therefore, we shall vary the reference solutions depending on the solutions we are trying to compare. For example, when

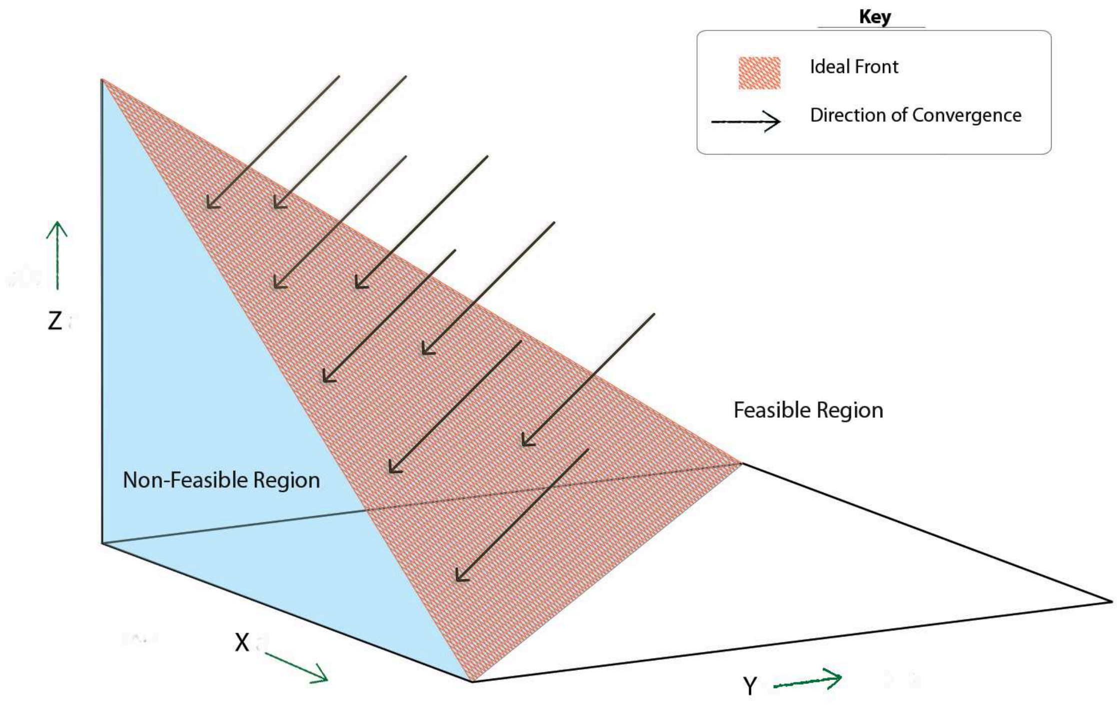


Figure 4.1: Direction of convergence towards the true front in a hypothetical and idealized minimization problem

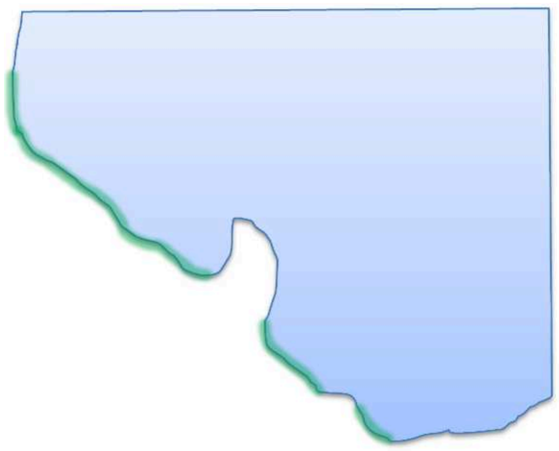
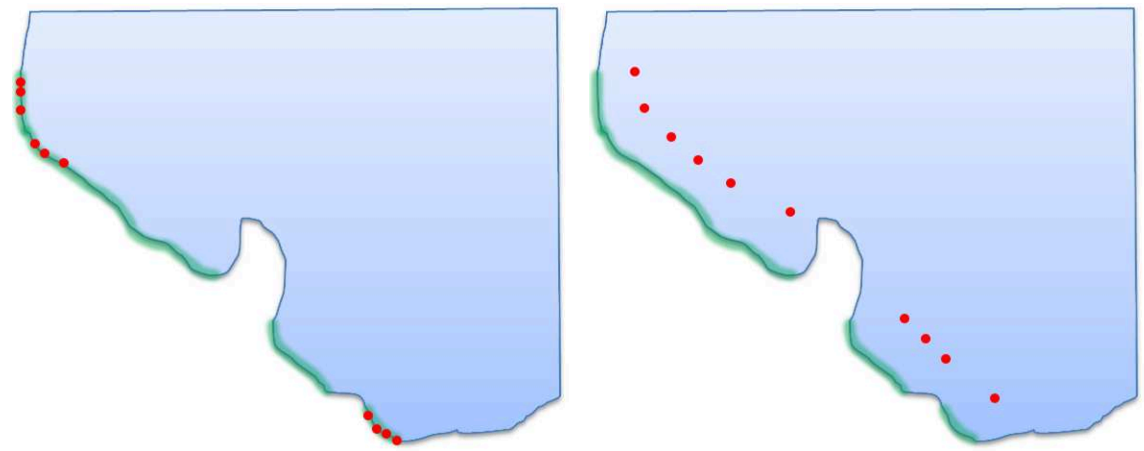


Figure 4.2: The design space of a minimization problem (blue) and it's convex, discontinuous Pareto front (green)

comparing fronts from two different generations of the same run, we use the front from the final generation as the reference solution. Furthermore, note that since each of the three objectives have different ranges, we normalize each objective in both fronts with respect to the extreme values of the objective in the reference front, before applying the metrics described below.



(a) The red points represent the Pareto front found. The front has a good distance but poor diversity

(b) The red points represent the Pareto front found. The diversity is good, but the distance is poor

We shall use Veldhuizen's Generational Distance (GD) metric [85] to quantify the distance between the obtained solution and a reference front:

$$\text{Let} \qquad d_i = \min_{k=1}^{|P^*|} \sqrt[p]{\sum_{m=1}^{\mathbb{M}} ({f_m}^{(i)} - {f^*}_m^{(k)})^p} \tag{4.1}$$

$$\text{Then,} \qquad GD = \frac{\left(\sum_{i=1}^{|Q|} {d_i}^p\right)^{1/p}}{|Q|} \tag{4.2}$$

Where Q is the solution under consideration; P^* is the reference Pareto-front; $\mathbb{M}$ is the total number of objective functions; and, ${f_m}^{(x)}$ and ${f^*}_m^{(x)}$ are the mth objective function values of the xth solutions of Q and P^* respectively. Note that since the different objectives are added to each other, they need to be normalized before use. A lower value of GD indicates a better solution, with solutions lying on the reference solution having GD=0.

For evaluating diversity, there exist Schott's [65] and Zitzler's [89] metrics. The former metric gives an estimate of the relative spacing between the points in the solution, considering a uniform spatial distribution as the ideal:

Schott's Metric: $$S_1 = \sqrt{\frac{1}{|Q|}\sum_{i=1}^{|Q|}(d_i - \bar{d})^2} \tag{4.3}$$

where, $$d_i = min_{k \in Q \wedge k \neq i} \sum_{m=1}^{\mathbb{M}} |{f_m}^i - f_m^k| \tag{4.4}$$

and, $$\bar{d} = \sum_{i=1}^{|Q|} \frac{d_i}{|Q|} \tag{4.5}$$

Here, $\bar{d}$ is the average of the distance measure d_i. Although it measures the relative spread of the solutions, this metric does not take into account the actual ranges obtained. Therefore, we need the 2nd metric:

Zitzler's Metric: $$S_2 = \sqrt{\sum_{m=1}^{\mathbb{M}} \left(\underset{i=1}{\overset{|Q|}{max}} {f_m}^{(i)} - \underset{i=1}{\overset{|Q|}{min}} {f_m}^{(i)} \right)^2} \tag{4.6}$$

This metric calculates the length of the diagonal of a hyperbox formed by the extreme function values observed. Therefore, it is a measure of the objective ranges but it does not take into account the solutions in between the extreme values; thereby necessitating the spread metric.

Deb et. al propose a single metric [18] for evaluating diversity that can be used in place of the above 2 metrics. We use this metric for our analysis:

$$\Delta = \frac{\sum_{m=1}^{\mathbb{M}} d_m^e + \sum_{i=1}^{|Q|} |d_i - \bar{d}|}{\sum_{m=1}^{\mathbb{M}} d_m^e + |Q|\bar{d}} \tag{4.7}$$

Where, d_i is the same distance metric described for GD, and $\bar{d}$ is their mean. Also, d_m^e is the distance between the extreme solutions of P^* and Q with respect to the m^th objective. However, this metric is closely tied to the true Pareto solution and needs many points in the true set to give an accurate picture. When we tried this metric for our data sets with a reference solution other than the true Pareto optimal solution, we found it is very sensitive to even the smallest changes in 1 parameter and gave very noisy results that are harder to compare (see Figure 4.4). Therefore, for some of the results, we needed to smooth the data across generations in order to see macro-trends – conclusions must be drawn compensating for this approximation.

Looking at equation (4.7), we find that it incorporates both the distances between points within a solution front, and the distance between the solution front and the

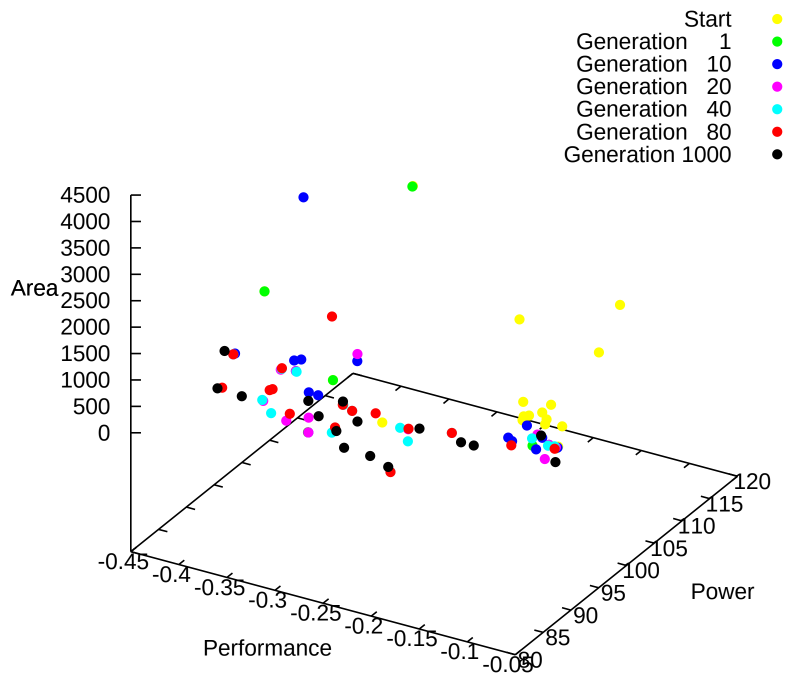


Figure 4.3: Pareto fronts found at various generations (Benchmark: Checksum, 10KB)

reference front. These two quantities are independent of each other and in our opinion, may not always accurately represent the situation when simply added together. Further study into this metric, or, proposing a new metric for comparing solution fronts (specially in the absence of reference fronts) can be an interesting topic for further study.

The sections in this chapter are grouped as follows: First, we describe the experiments to improve the performance of the tool. These consist of setting appropriate number of generations and choosing a crossover probability. Then, we present the front obtained by running all the benchmarks and a single workload size. We will analyze all subsequent results with respect to this solution. This is followed by experiments for improving the results obtained – Changing workload sizes, and an analysis of the benchmarks. We then demonstrate the tool's modularity by adding an additional requirement – that of real-time performance. This also validates the tool to an extent. Finally, as a case study, we analyze the actual processor configurations found in the context of real applications.

Unless otherwise stated, the workload is EMGII with an input data size of 10KB.

4.1 Selecting Number of Generations

As mentioned above, genetic algorithms start with an approximation ($\mathbb{P}'$) of the true solution ($\mathbb{P}$). In each generation, the algorithm attempts to improve $\mathbb{P}'$ so that the

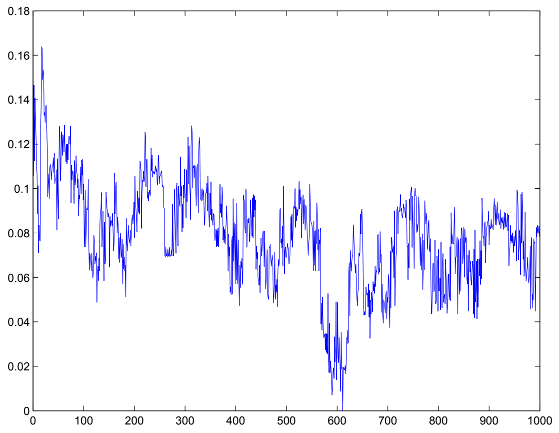


Figure 4.4: The distance metric fluctuates widely

error between $\mathbb{P}$ and $\mathbb{P}'$ becomes smaller. Therefore, the more the generations we allow the algorithm, the more confident we can be of being 'close enough' to the true front. However, more generations means a longer run-time of the algorithm. Hence, we need to trade off (possible) accuracy of the solutions for efficiency.

In order to see the relation between accuracy of the solution and number of generations, we ran the fastest benchmark, *checksum*, for 1000 generations. Although our goal is to optimize across all the benchmarks, checksum being the fastest gives us an indication of the number of generations required without being prohibitively expensive to run.

Figure 4.3 shows the solutions found at some of the early generations of algorithm, as well as the last solution found - at generation 1000. The performance, power and area units, are $-1 * IPC$, average power in mW, and approximated area in mm^2 respectively. We see that the algorithm starts with a random collection of points, and as the generations progress, the shape of the front increasingly resembles that of the final front. At generation 80, the solution points already have a good range and are very near the 1000th front. Also, the difference between the starting generations and the 80th generation is much more than that between the 80th and the 1000th generation. Around this stage, the algorithm starts concentrating on increasing diversity in the solutions found – i.e., a more uniform spread along the Pareto front. Let us consider a generation a little further on – a closer comparison between the 100th and 1000th generations can be seen in Figure 4.5. As can be seen, the 100th generation continues the trend seen in the previous generations. It takes only 10% the time taken by the 1000th generation and still manages to be very close to the final result.

In order to see the above observations quantitatively, we calculated the distance and diversity metrics as given in equation (4.2) and (4.7) respectively. Figure 4.6 shows these values calculated for the entire run of the algorithm; keeping combined Pareto solutions

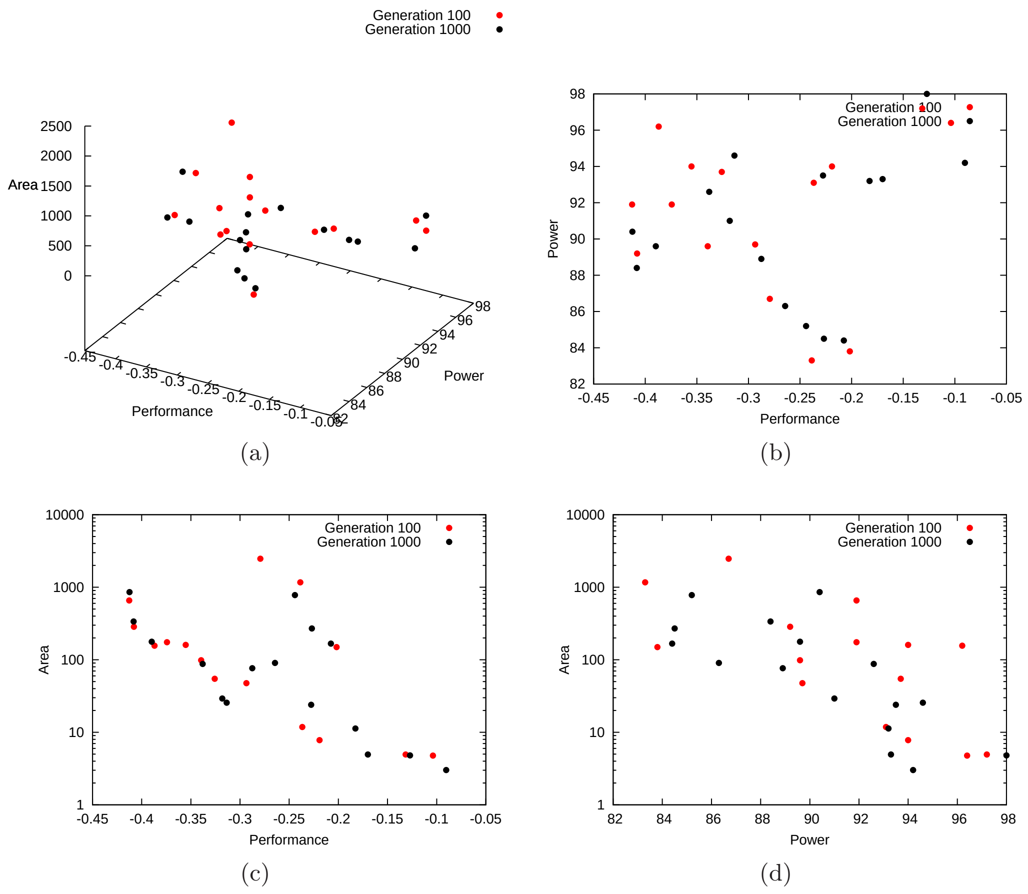


Figure 4.5: Comparison between solution frotns from the 100th and 1000th generations (Benchmark:Checksum)

from many runs as the reference front. We found both metrics to be very noisy, specially the diversity metric (see figure 4.4). Therefore, to make them easier to compare, we smooth the data using a moving average with a span of 20 generations.

We find that the distance metric (GD) declines rapidly until about the 100th generation, after which it fluctuates around GD=0.4 for a long time until finally dropping to zero around generation 600. The rapid decline is of course due to the solution fronts converging towards the reference. After generation 51, it can be seen that the general trend is that as spread decreases, distance increases and vice versa. The behavior from generations 100-593 are also expected - as the algorithm searches for new solutions, the distance fluctuates. A minor rise in both GD and spread may not mean that the front is actually further away than the one previously – since we have finite points in the reference front, points that are the same distance from the *actual* front may give slightly varying distance values from the reference front. The behavior at the tail end reflects the fact that the reference solution is a combination of several solutions, and therefore also contains the final solution of the run in question. The apparent

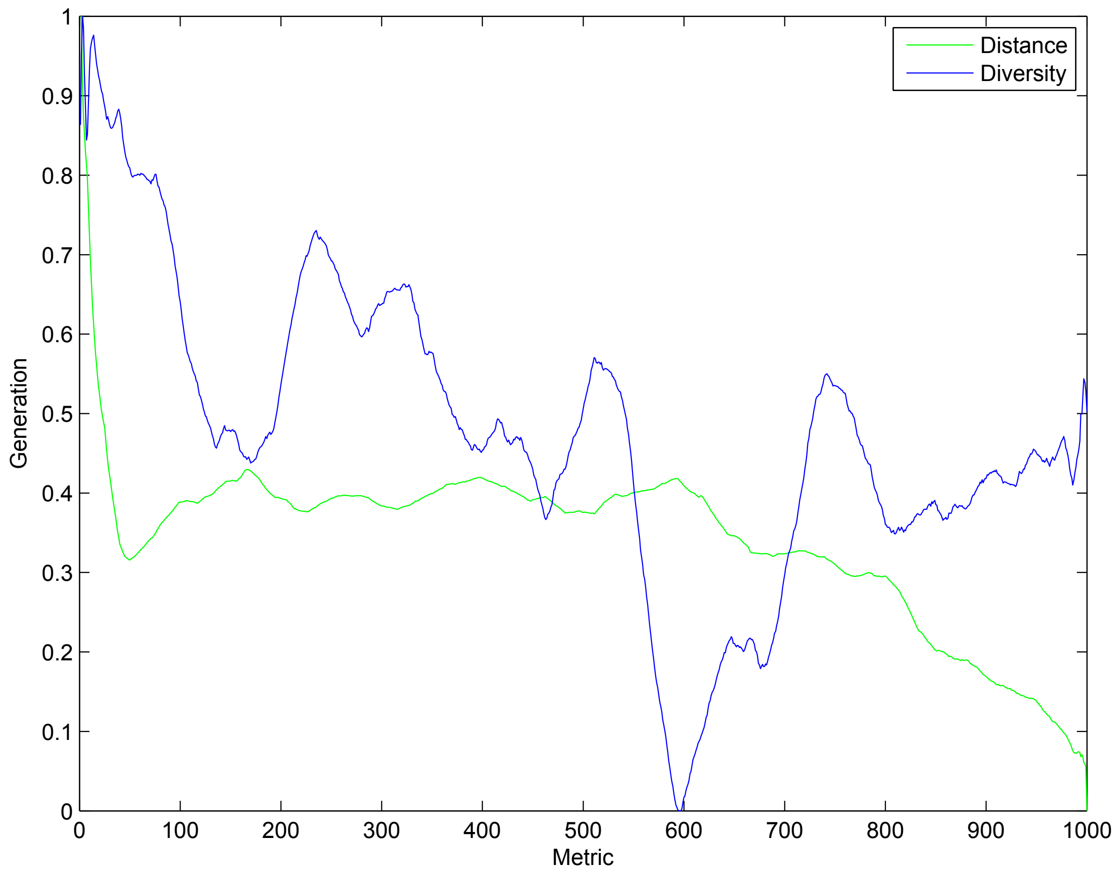


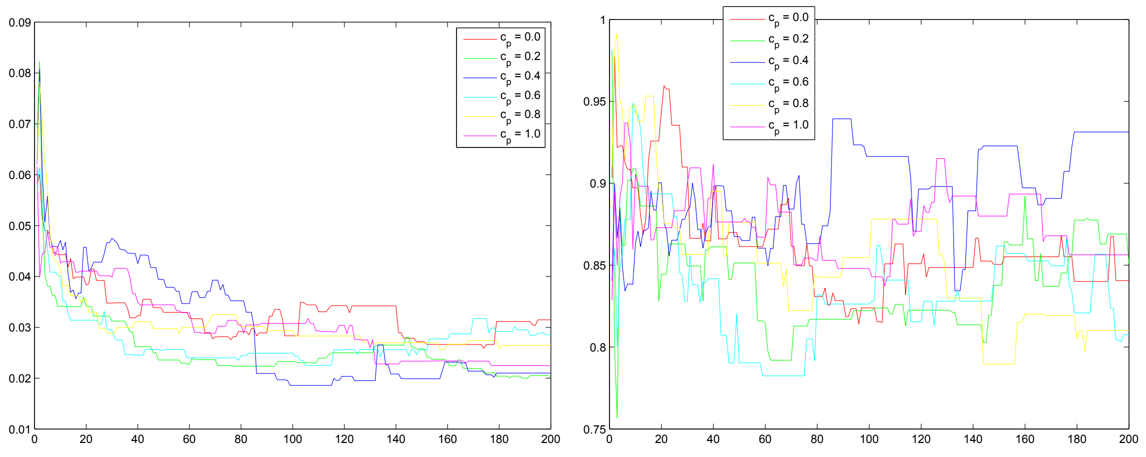
Figure 4.6: Smoothed distance and diversity metrics over 1000 generations (Benchmark: checksum)

rapid convergence seen therefore, is in fact the algorithm moving towards its own final solution – a foregone conclusion. Therefore, we do not consider this region in our analysis.

Since after generation 100, the algorithm seems to oscillate between improving spread and distance at the cost of the other; without loss of precision in both quantities at the same time, we can stop the algorithm around generation 100. Since GAs are random by nature, in order to be sure of getting good results, we run every subsequent experiment for twice this time, i.e. – 200 generations. This also compensates for the smoothing we performed.

4.2 Selecting Crossover Probability

Selecting a good value for crossover probability (p_c) leads to a faster convergence of the algorithm. The mutation probability was fixed to $p_m = 1/n = 0.0769$ as decided in section 3.8 [p. 31]. The crossover probability was varied from 0 (never perform crossover) to 1 (always cross parent genes) with a step size of 0.2 in order to examine the effects of



(a) Distance metric (b) Diversity metric

Figure 4.7: Distance and Diversity metrics for various crossover probabilities (c_p) (Benchmark:checksum)

different crossover probabilities on the final result. In this section, we discuss the result of this experiment and select a value of p_c for the subsequent runs of the algorithm.

Note that the aim is to select a crossover probability such that the algorithm converges faster towards the ideal front. Since running our original problem with all the benchmarks is prohibitively time consuming, we run the crossover experiment on a subset of the original problem – with only a single benchmark (Checksum), and use the value of crossover probability suggested by this experiment for solving the bigger problem. Although this might not give the *optimal* crossover probability for the original problem, it will give a value that is *good enough*. As genetic algorithms are by nature robust to parameter selection, and work well enough with "good enough" values, this approximation is not likely to affect the result much. Checksum was chosen for the experiment as it is the fastest running benchmark of all the benchmarks. In addition, the population size was fixed at 20 individuals, and for each value of p_c, the algorithm was run for 200 generations.

We found that it is harder to visibly inspect the fronts found to see which probability evolves fronts faster [1]. Therefore, we created a combined Pareto front from all the crossover probabilities. We then computed the GD and spread metrics with respect to this combined front for each generation of the algorithm. Figure 4.7 shows the two metrics. Note that these are the un-normalized, un-smoothed metrics. They appear less noisy due to the fact that the number of generations plotted is much lower. We see from the graphs that $cp = 0.2$ and $cp = 0.6$ both seem to perform the best in terms of distance and spread. Therefore, we choose a crossover probability of 0.6.

[1]Nevertheless, actual comparison fronts can be found in our paper[15]

4.3 Combined Optimization for ImpBench

With the setup tuned as above, we run an exploration to find optimal processor configurations for the 8 benchmarks in the original ImpBench suite (refer Chapter 2). Each individual is evaluated against all benchmarks, and the algorithm optimizes the average performance, power and area across these benchmarks. Assuming the benchmarks cover the application spectrum, optimizing for the average of the individual benchmark's metrics ensures that we get processor configurations that are expected to work well with any real life application.

Figure 4.8 shows the Pareto front found. Although we optimize $-1 * IPC$, the figure reports IPC as this is the actual measure. The area axis are shown on a logarithmic scale because it varies widely, and we want to emphasize the points found at the lower end of the axis. Each point is a different processor configuration, and is labeled identically in each of the views.

The various processor configurations corresponding to the above figures are shown in Figure 4.10[2]. Given the objective trade offs and the corresponding processor configurations, appropriate design points can be chosen. A case study on selecting processor configurations from this data by considering real implant applications is presented in Section 4.7. We shall also use this Pareto set as the baseline with which all subsequent results will be compared.

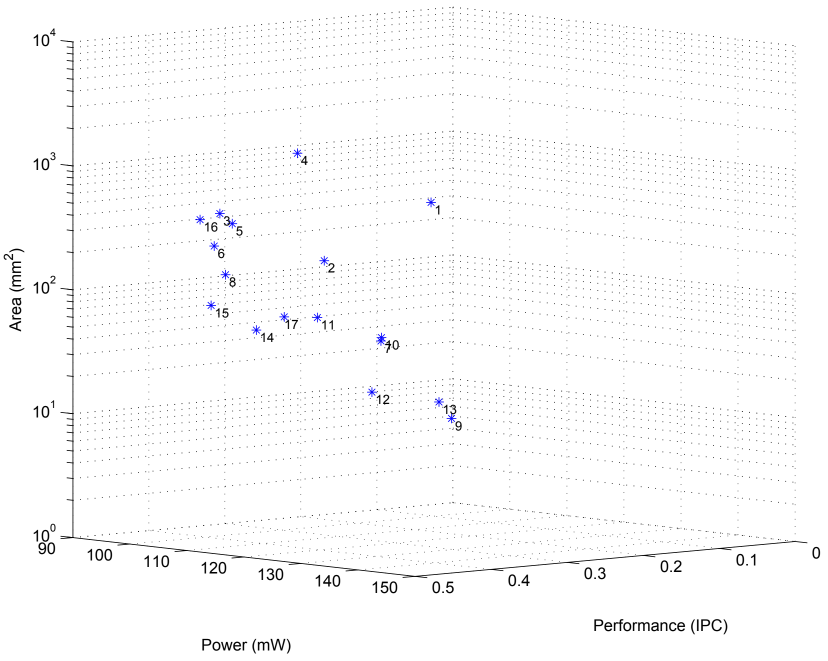
4.4 Impact of Workload Size

By default, we ran our experiments with 10KB of input data. In order to see the impact of changing the data size, we also ran some experiments with 1KB of input data. Moreover, if we find that the two data sizes do not differ much in terms of output ranges, future experiments can be run with 1KB data without much loss of precision, and with a reduced runtime – while 10KB data takes on an average 541 seconds to evaluate a single individual for all benchmarks on our machines, the 1KB data takes 349 seconds, a 35% improvement.

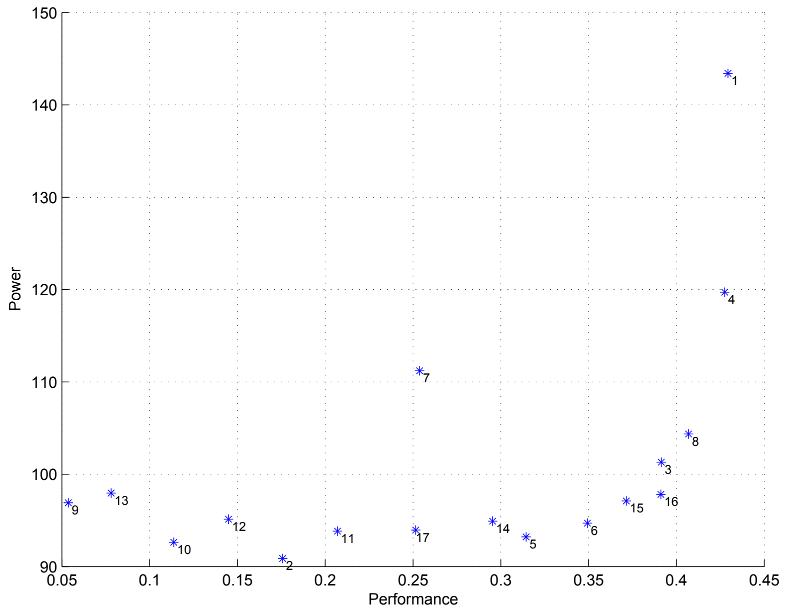
Figure 4.9 shows the comparison between the fronts evolved using the two different data sizes. We see that while the shape of the 1KB front follows that of 10KB, the 1KB processor configurations are in fact 'worse' – in general, they have higher power, more area, and lower performance. For example, consider Figure 4.9b. For IPC falling between 0.05 to 0.3, The power and performance figures are nearly identical for the two data sizes. However, for relatively higher performances, i.e, $IPC > 0.3$, the 10KB data size is able to deliver better performance in terms of power. Similarly, for the other pairs of objectives, the 10KB points in general appear to dominate the 1KB points. We also see similar trends for these two data sizes in Section 4.5

We believe the above behaviour may be due to the fact that the 10KB data size is

[2]Note: Bimodal size, BTB sets and BTB associativity are only valid for branch type of Bimodal



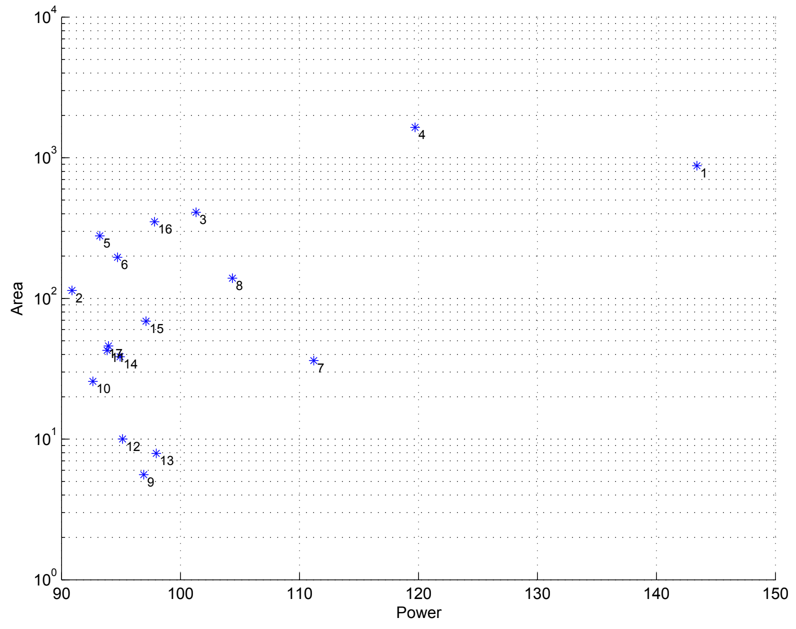
(a)



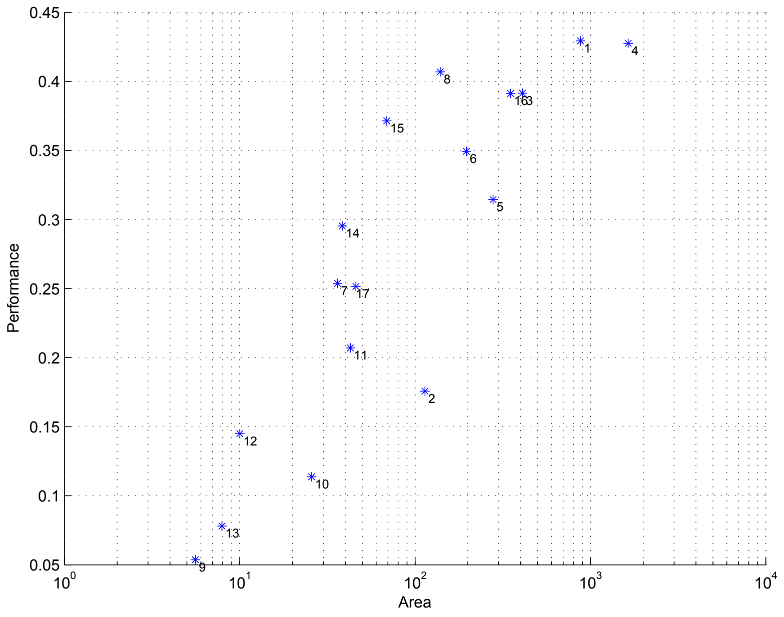
(b)

Figure 4.8: Pareto front optimizing all benchmarks

better able to fill the cache and branch tables, and thereby has fewer processor stalls, hence giving better performance for the same power/area and vice versa. As a future study, it would be interesting to study more input data sizes.



(c)



(d)

Figure 4.8: Pareto front optimizing all benchmarks

4.5 Extension with Real Time Constraints

As this was one of the first steps towards designing an implant processor, we wanted to make sure that the framework was extensible, in order to facilitate the addition of new domain specific information into the framework as it gets available. In order to test this property of the framework, we devised a synthetic implant application with a

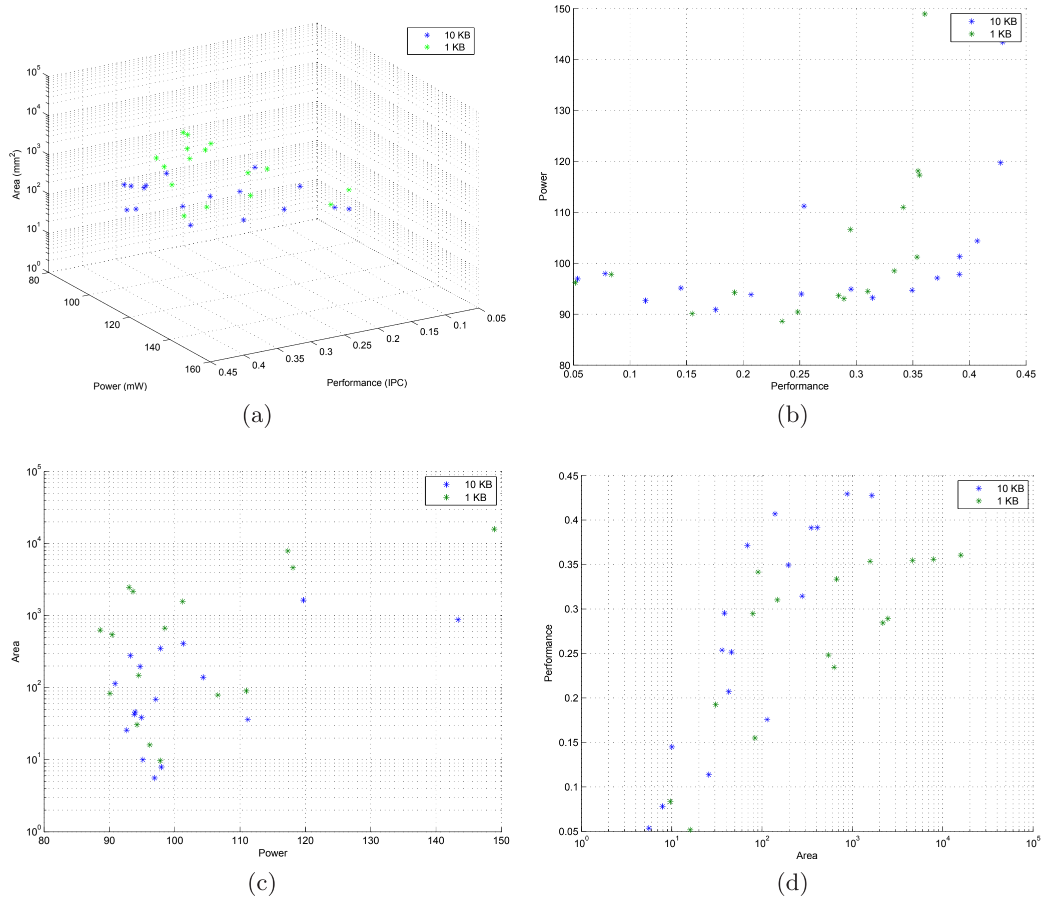
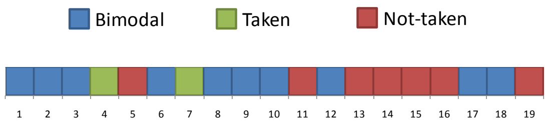


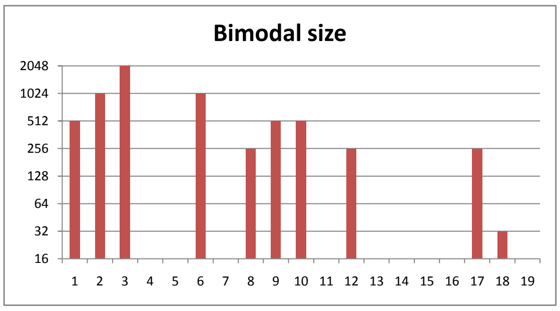
Figure 4.9: Pareto fronts evolved with workload sizes of 10 KB and 1 KB

hard realtime deadline. Indeed, many implant applications have realtime requirements so formulating a synthetic problem with such a constraint does not fall far from practice. We used a modified version of the DMU and Motion benchmarks (called StressDMU and Stress-Motion respectively)[16], to represent a single iteration of these (repetitive) applications. Further, this iteration was chosen to be the worst-case iteration (in terms of instruction-cycle count) for each of the two original benchmarks. Combined with data-integrity checks, compression and encryption, the stress benchmarks represent an atomic action for an implant application from data collecting, processing, to transferring. In a real application, this atomic action must be finished, for example, before the next set of input arrives. Therefore, we constrain the total time required for this combined operation as a hard realtime deadline. We use 1KB data input sizes, because, as we studied before, they run faster while giving the same general trends as the 10KB data – except for a smaller performance distribution (at the higher end). Since realtime deadlines cutoff performance at the lower end, this is not a problem.

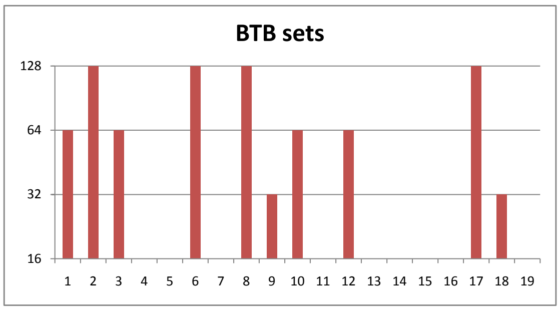
We started with a relaxed initial deadline of 60 secondsFigure 4.11. We notice that the front evolved with this deadline has almost the same ranges as the one without any



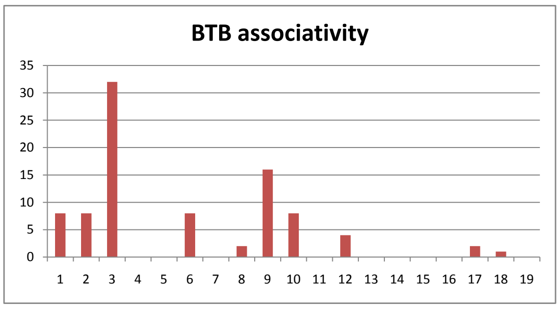
(a) Branch Prediction Type



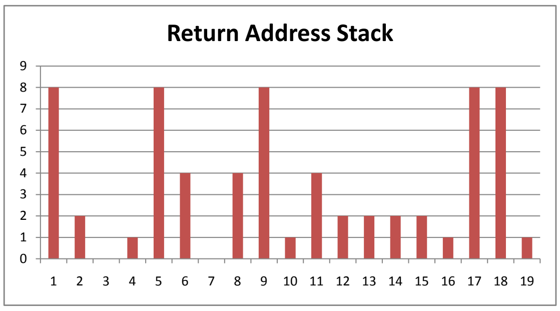
(b)



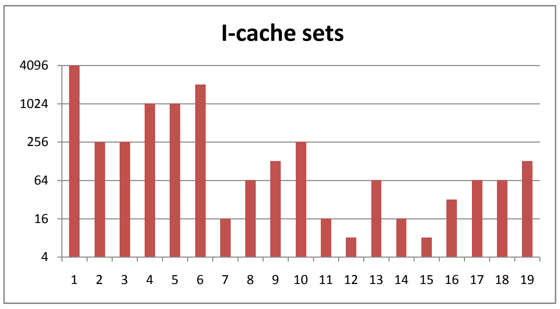
(c)



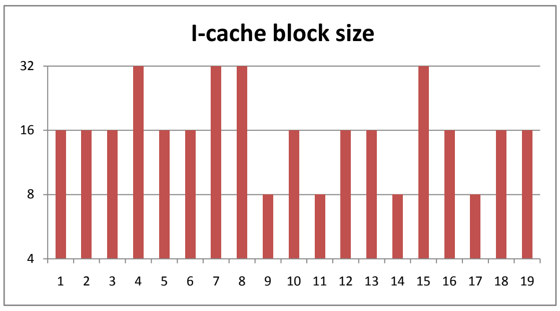
(d)



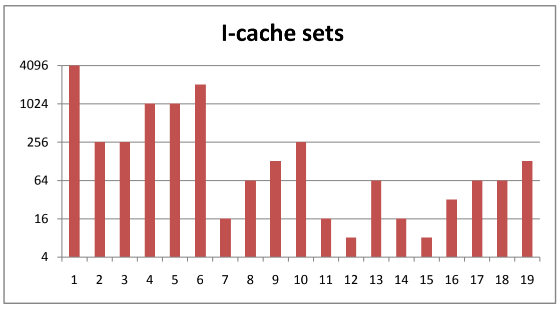
(e)



(f)

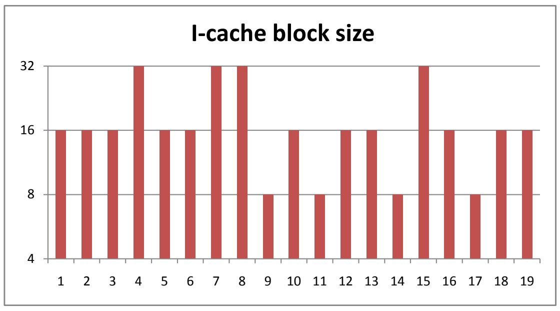


(g)

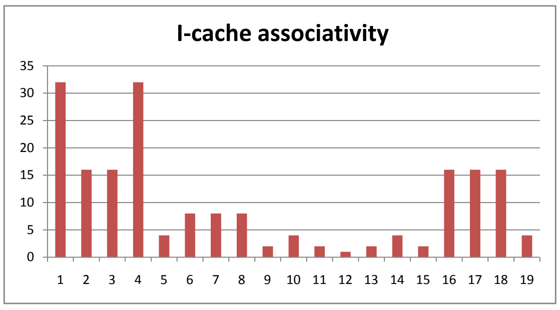


(h)

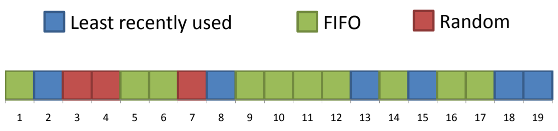
Figure 4.10: Parameters of the Pareto optimal processor configuration



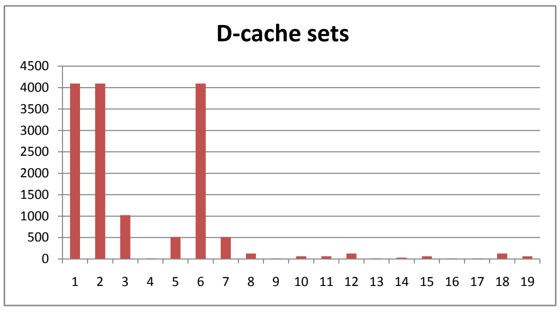
(i)



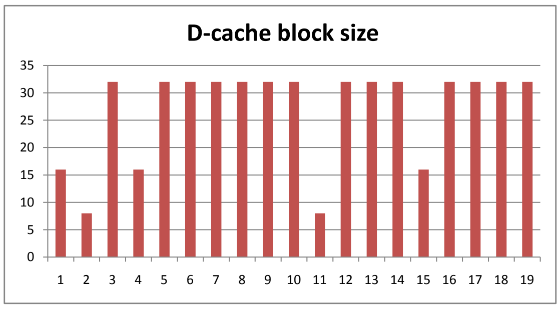
(j)



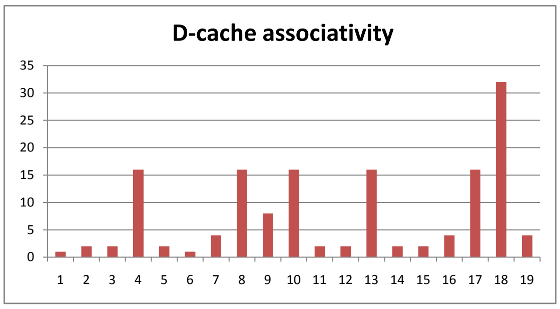
(k) I-Cache replacement policy



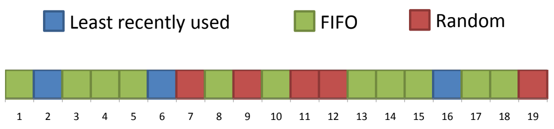
(l)



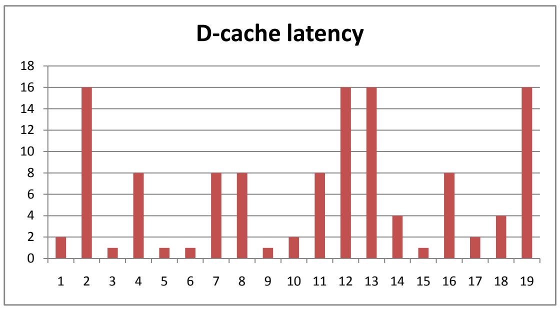
(m)



(n)



(o)



(p) D-Cache replacement policy

Figure 4.10: Parameters of the Pareto optimal processor configuration

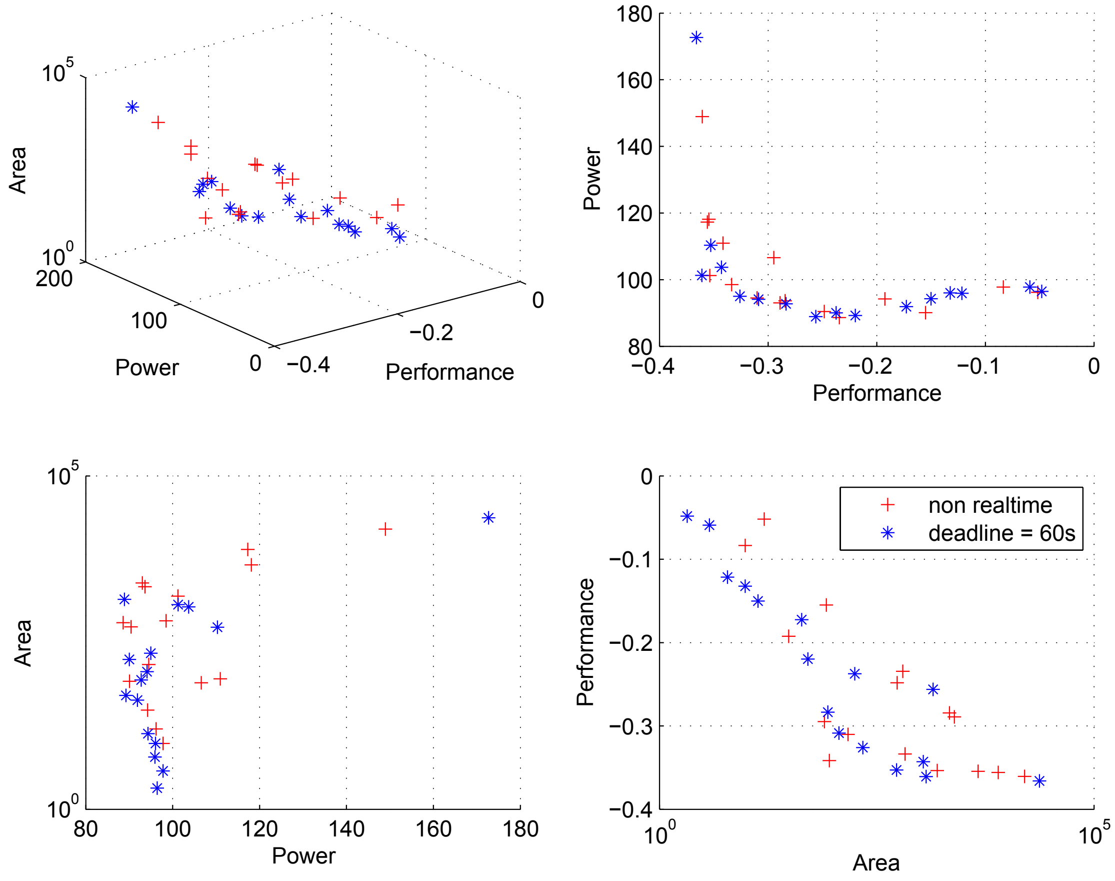


Figure 4.11: Fronts created with and without realtime deadline. This one has a very relaxed deadline, therefore, not much change can be seen in terms of ranges.

deadlines, except perhaps being more defined (possibly because the search space has been reduced). We then ran successive experiments with tighter deadlines. We observed that sharp cutoffs could be seen at the lower end of performance with tighter deadlines. In Figure 4.12 we show two of the strictest deadlines to showcase the cutoffs. This behaviour is exactly as expected - with tighter deadlines, only those processors that can deliver higher performance to the applications remain feasible.

4.6 Comparative study of the benchmarks

So far in this work, we have used the benchmarks to rate processor configurations. We propose that we can also use our tool to characterize the benchmarks themselves. The idea is that each benchmark, with its own power and performance behaviour, and different ways of utilizing the processor components, will lead to different fronts. Based on the comparison between benchmark fronts, we can draw inferences on what processor configurations the benchmarks favour, and how applicable a benchmark would be in a given scenario as compared to another from the suite. Here, we shall present

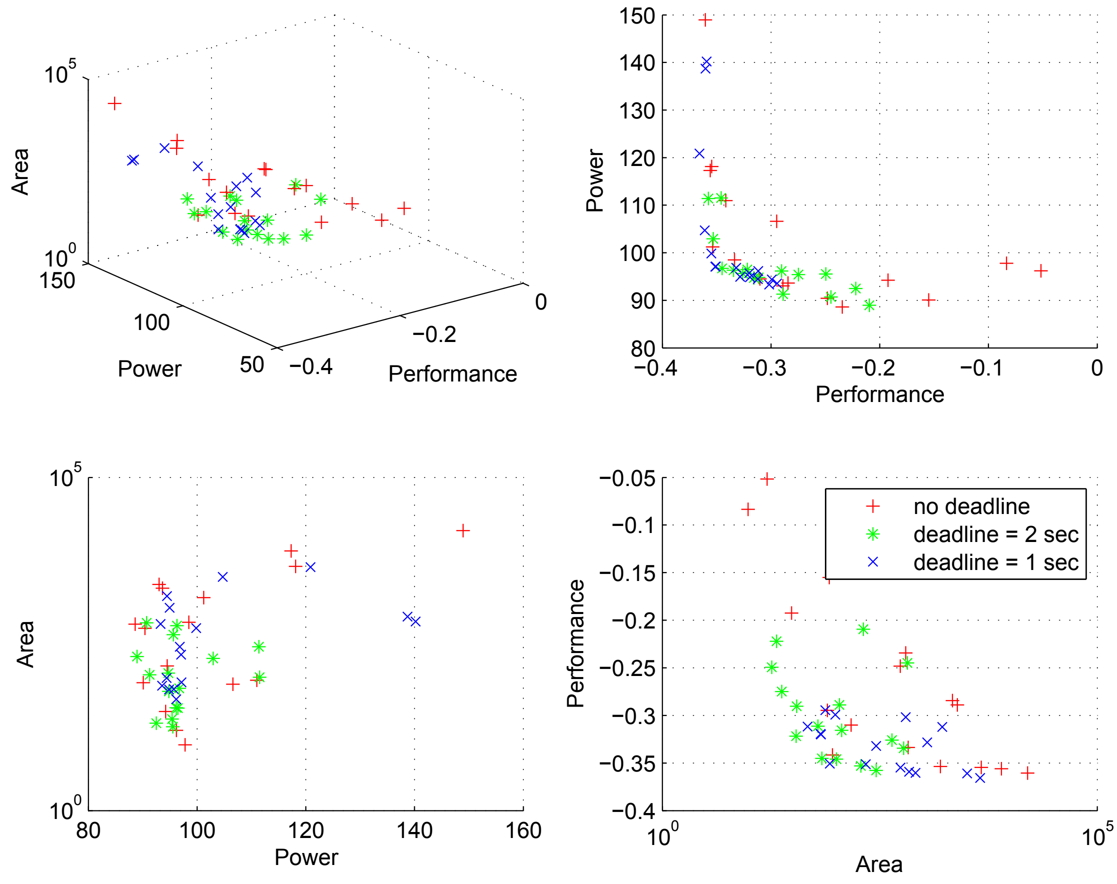


Figure 4.12: As deadlines become tighter, more cutoff can be seen at the power end. This is clearest in the Power-Performance graph

an overview our findings. A more in-depth study, including comparisons of the actual hardware configurations evolved, can be found in [79].

Figure 4.13 to 4.15[3] show such fronts found for each of the benchmarks, with the benchmarks grouped by category (encryption/data integrity/compression/real applications/stress). Table 4.1 shows the relative distances and spreads of these fronts from the standard front of Section 4.3. We refer to the latter as the *all* or *average* case (as it evolves to optimize the average figures from all benchmarks).

Analysis the above graphs and table has revealed new information with respect to the coverage of the design space and the hardware implications contributed by each ImpBench component. Results indicate that, from the lossless-compression benchmarks, *"mlzo"* provides better *GD* and similar Δ to *"fin"*. On top of this, it also features faster simulation times (about $\times 3$ faster, according to Table 4.1). From the

[3]Note that the performance metric in the axes is CPI, the inverse of IPC – in order to be consistent with [79]

Benchmark	*GD*	**Δ (normalized)**	**Sim. time* (average) (sec)**
miniLZO	0.082	0.383	3.07
Finnish	0.115	0.367	21.82
MISTY1	0.083	0.181	16.65
RC6	0.049	0.151	9.37
checksum	0.090	0.189	0.80
CRC32	0.111	0.285	5.46
motion	0.094	0.163	31.16
DMU4	0.085	0.174	37.25
stressmotion	0.088	0.266	1.33
stressDMU3	0.105	0.143	0.60
all	**0.000**	**0.000**	**125.58**

Table 4.1: Pareto-front distance and normalized-spread metrics and average simulation time per benchmark.

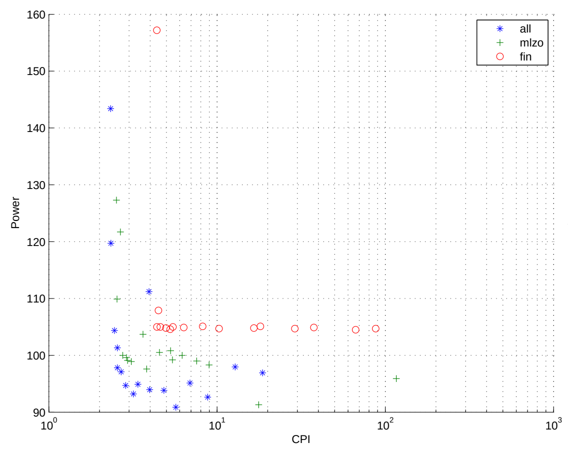
symmetric-encryption benchmarks, *"rc6"* displays excellent characteristics, namely the smallest *GD* and the second best Δ, meaning that it traces the aggregate Pareto front with high fidelity. According to Table 4.1, it is also about twice as fast as *"misty"*. Of the data-integrity benchmarks, *"checksum"* performs consistently better than *"crc32"* and features the overall shortest simulation time (0.80 *sec*) across all full benchmarks.

As for the synthetic-benchmarks case, no single benchmark could be unanimously ranked higher due, mainly, to the complex nature of the results. This actually shows the usefulness of both real benchmarks which, also, feature similar simulation times (and the largest in the whole suite). Also, looking at the new stressmarks, we can see that although they display some variability in predicted processor specifications w.r.t. the full synthetic benchmarks, they both track the true Pareto front closely. Careful use of the stressmarks can seriously reduce simulation times up to ×30, which is an impressive speedup and a good tradeoff between DSE speed and accuracy.

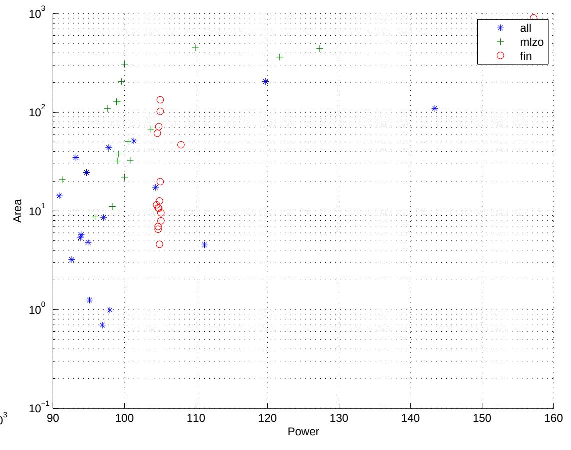
4.7 Case Study with Real Applications

We have already listed the specifications of a few real implant applications in Chapter 2, Table 2.2. In this section, we briefly show how our design points compare to these applications. A more detailed study is left as future work. We use the baseline results of Section 4.3 for our study. However, the baseline optimizes the average of the benchmarks run. In order to make a comparison, we ran the processors in the baseline set on the same 'conceptual SiMS application' as was used for the realtime constraints – a combination of miniLZO, rc6, checksum, stressdmu3 and stressmotion. We find the *combined* behaviour of these benchmarks and use this for comparison.

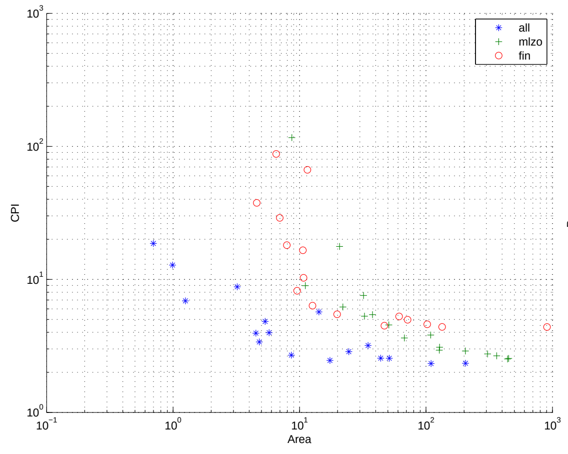
Since the performance metric of the applications in Table 2.2 is in terms of single loop execution time, we use the time taken by the simulated processor to run our



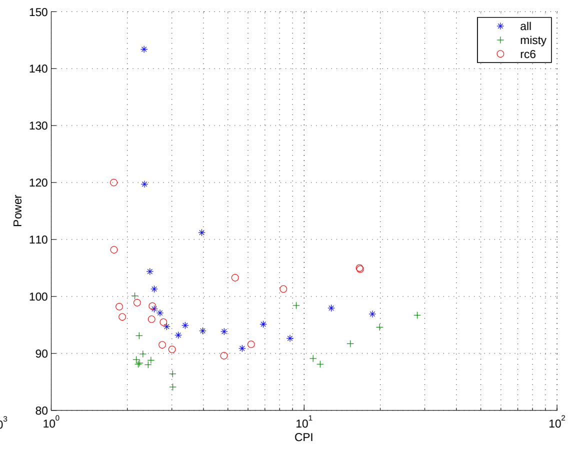
(a) Compression (Performance – Power)



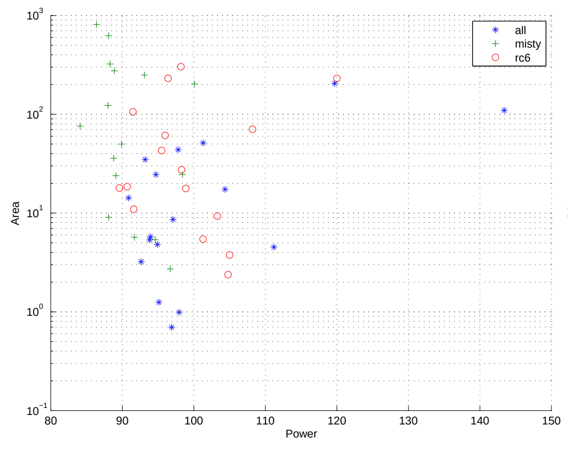
(b) Compression (Power – Area)



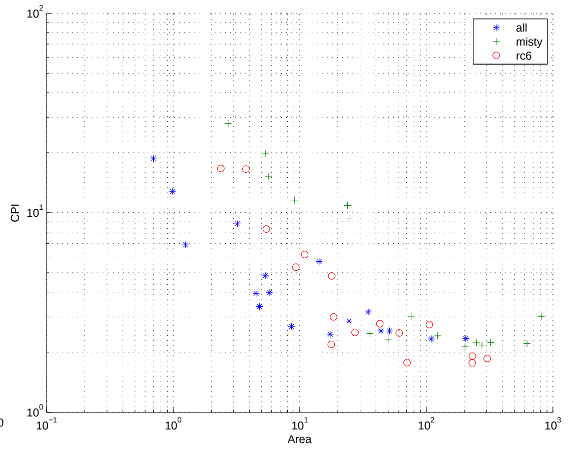
(c) Compression (Area – Performance)



(d) Encryption (Performance – Power)

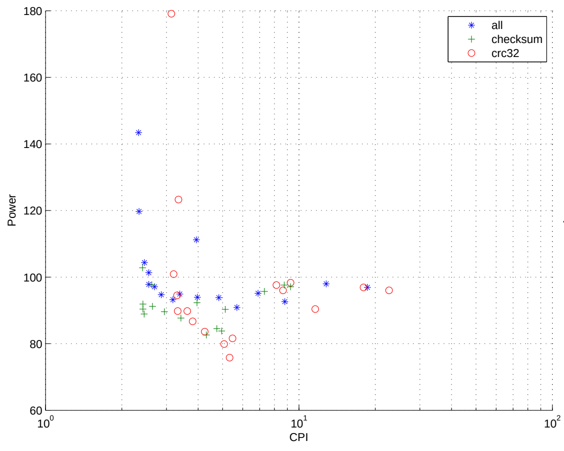


(e) Encryption (Power – Area)

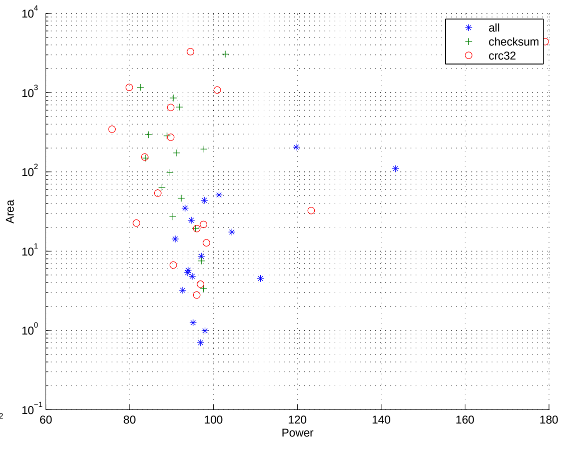


(f) Encryption (Area – Performance)

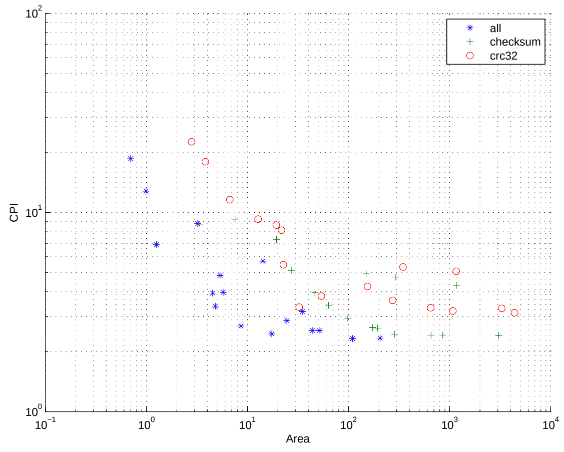
Figure 4.13: Comparison of benchmark pairs - I (Compression, Encryption)



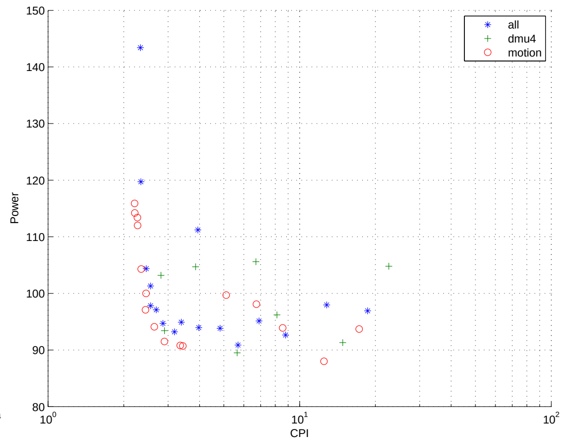
(a) Data Integrity (Performance – Power)



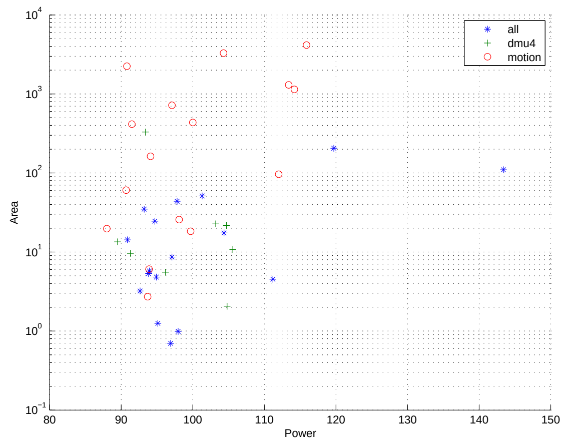
(b) Data Integrity (Power – Area)



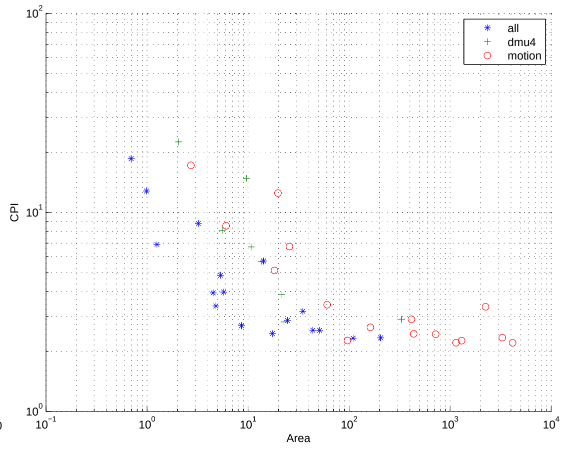
(c) Data Integrity (Area – Performance)



(d) Real Applications (Performance – Power)

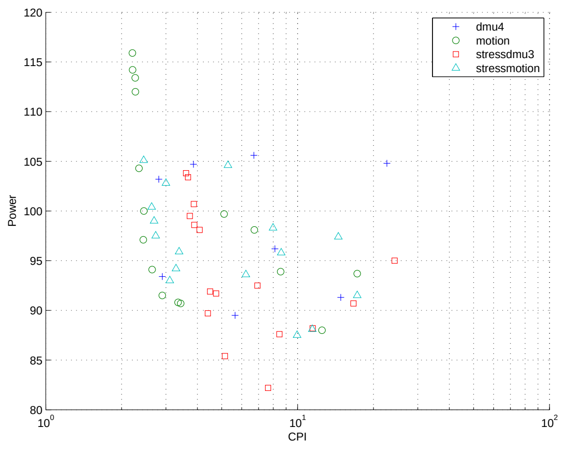


(e) Real Applications (Power – Area)

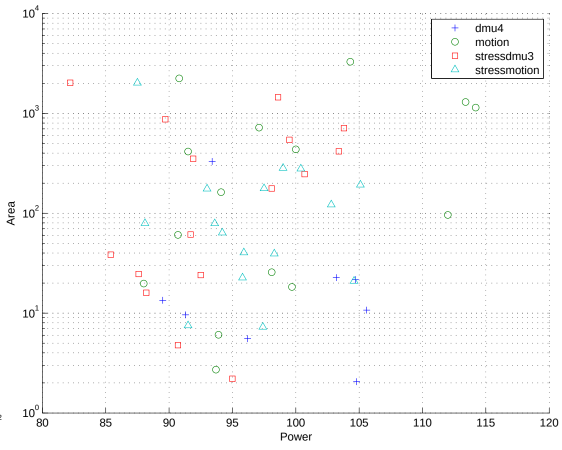


(f) Real Applications (Area – Performance)

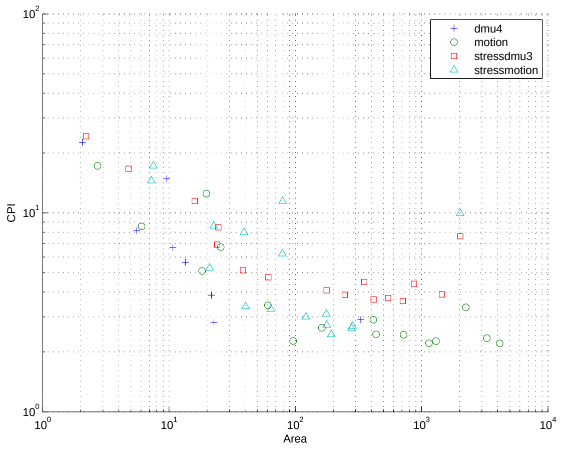
Figure 4.14: Comparison of benchmark pairs - II (Data Integrity, Real Applications)



(a) Stress Benchmarks (Performance – Power)



(b) Stress Benchmarks (Power – Area)

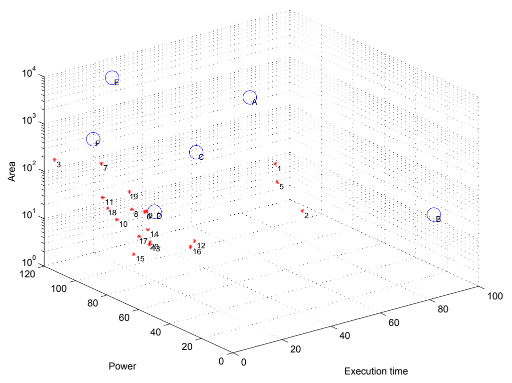


(c) Stress Benchmarks (Area – Performance)

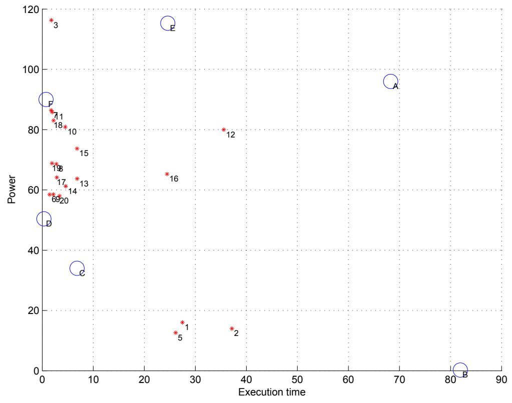
Figure 4.15: Comparison of benchmark pairs - III (Stress Benchmarks)

conceptual application as the single loop execution time. Figure 4.16 shows our design points and the case study points, where the case study points are labeled according to the numbers in the table. From such graphs, we can design generic processors that can cover most, if not all of the studied applications.

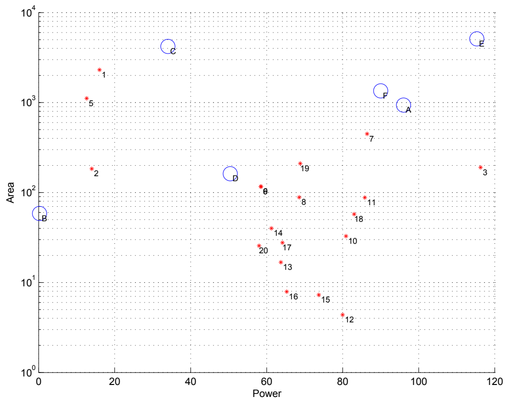
Let us make a short analysis. We notice that configuration #A and #E are dominated by most of our processor points in all three dimensions. Therefore, we can easily replace this application by one of our processors. We found that #B is dominated in terms of execution time and area but not in terms of power. Looking at the application [23], we see that #B is connected to an external power source. Therefore, power is not an issue for this application, and any of the processors that dominate it may replace it. Furthermore, we see that #D and #F are both over designed on performance - the actual applications are gastric pressure monitoring and monitoring



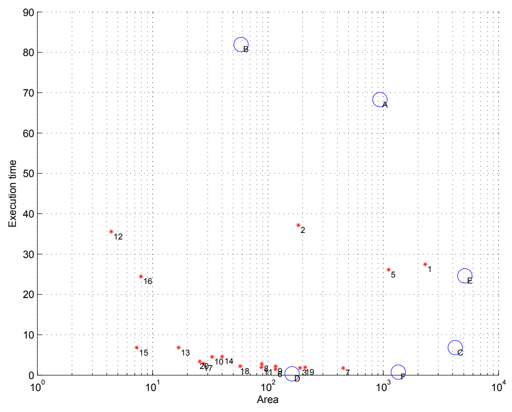
(a) 3-Dimensional



(b) Time-Power



(c) Power-Area



(d) Area-Time

Figure 4.16: The objective space for our designs and the real implants

of AEG, which do not change very rapidly. Therefore, in our opinion, processor points with slightly higher execution times can also safely cover these applications. As for point #C, we find that no processor points in our design space can deliver the high performance needed by the application. However, looking at the area and power axes, we see that #C has considerably more area margin than our designs and more power than #1, #5 and #2. Therefore, if our hypothetical SiMS processor were to be used for application #C, we could use the extra power and area budget for an extra hardware accelerator to deliver the performance required. Keeping this analysis in mind, we can use processor design #1 or #5 for applications #A, #B, #C, #D and #E. As for application #F, we can use #7, which can also be used for #A, #B, and #E. Therefore, we can develop these 2 processor configurations as the generic processors we were searching for. However, these results are considering the XTREM simulator which is based on the XScale processor family. We expect a much better behaviour in future when we will use processors more suitable to low power embedded applications.

Future Work and Conclusions 5

5.1 Conclusions

In this work, we presented a framework for multi-objective design space exploration of implantable processors. This section, we highlight the major features of the framework and summarize the results obtained from using it.

5.1.1 Framework

Multi-objective design space exploration can help designers in making intelligent choices about processor design by making available the Pareto optimal set of processor configurations. In our case, the optimality criteria was based on minimizing power and area and maximizing performance. We saw that genetic algorithms are best suited for such design space exploration, and we chose to use a popular state-of-the-art multi-objective genetic algorithm (NSGA-II). This algorithm is supported by a cycle-accurate simulator (XTREM) and a cache-simulator (CACTI), which together simulate the three objectives. Interface scripts connect the algorithm and simulators. The major features of the framework are summarized as follows:

Parallel The framework was parallelized using RPC, with a single "client" running the evolutionary part, and multiple "servers" for the computation-intensive simulation.

Modular The client-server model and the supporting script interfaces ensure that we need only change the interfacing script in order to change the simulator.

Configurable The framework is highly tunable, and reads all of its properties through simple configuration files. This includes both the parallelization properties (list of allowed machines) and genetic algorithm properties such as the number pf optimization variables, their size and ranges.

Scalable The above two properties also make the framework scale very easily with the number of computing resources available. With the population size of 20, the speedup is almost $\lceil 20/N \rceil$ with N machines available for use.

Adaptive and Reliable The network is periodically checked for free machines out of the list of possible machines. The framework also periodically tries to start new server processes in idle machines. In case a machine fails midway in simulating a given individual, re-computation is attempted on another machine. In order to prevent possibly infinite re-computations due to bad processor configurations that themselves cause the server to fail, the number of times re-computation is attempted is fixed to a pre-configurable number (2 in our case). In order to prevent network related losses, the TCP protocol is used for data transfer.

5.1.2 Experimental Results

After designing and implementing the framework, we used it for various experiments. These are summarized as follows:

Selecting number of generations We introduced the distance and diversity metrics which have been proposed in literature for comparing the ideal Pareto front and solution fronts from using multi-objective genetic algorithms. However, we use these metrics to analyze the quality of the solutions found (in a reduced problem) over the generations, in order to determine how long we need to run the algorithm for (for the full problem). We found that the algorithm solution front rapidly approaches the Pareto front until generation 100 after which the front stabilizes and does not seem to improve one metric much without degrading the other. Therefore, we select the number of generations equal to 200 as a conservative measure. As far as we know, these metrics and such an analysis has not so far been used in order to set the number of generations.

Selecting crossover probability The crossover probability was set by comparing the diversity and distance metrics of the fronts found by various crossover probabilities with a combined front (approximating the ideal front) found by running many instances of the (reduced) problem. We believe such an analysis is also unique to this work.

Changing workload size We experimented with changing the workload size from the default of 10KB to a lower size of 1KB, in order to see if we could speedup the framework by using the lower size. We found that the larger workload gave a higher performance for the same power/area as the smaller size. Since the characteristics were noticeably different, we continued to work with 10KB data.

Addition of realtime constraints Realtime constraints were added to the framework as a constraint violation in the genetic algorithm; and formulated as the maximum time allowed to finish an envisioned task consisting of one encryption, one compression, and one data-integrity benchmark along with two real-world stress-benchmarks. Starting with a time-constraint of 60 seconds, where we found was met by almost all benchmark configurations; the constraint time was incrementally lowered. At violation times of 2 seconds and 1 seconds each, we see clear cut-offs (as compared to no-constraint run) at the lower performance end, with a more severe cutoff for 1 second than for 2 second deadline.

Characterization of benchmarks We used the framework for studying the benchmarks in the extended ImpBench suite. Separate Pareto fronts were independantly evolved for each of the benchmarks. These 'benchmark' fronts were presented in relation to the front evolved through the optimization of the combination of all original ImpBench benchmarks ('average front'). Designers can use such a characterization in order to choose which benchmarks to run. For example, it can be seen that stressDMU3 has very competitive spread and distance metrics while having the lowest run-time of all the benchmarks. Therefore, if a designer were to use only stressDMU3 instead of the entire suite, he would save 99.5% of the runtime while maintaining comparable accuracy in the fronts obtained.

Choosing generic processors Finally, we compared 6 real-life implant applications to our Pareto front. By looking at the points in the Pareto optimal front that dominate the real-life implants, and by making case-specific design allowances (for example in one application, power is not an objective), we propose a set of 2 processors that between themselves can meet the requirements all these real-world implant scenarios.

5.2 Future Work

Throughout this work, an attempt has been made to identify improvements that can be made, or further experiments that can be done. In this section, we list some of these grouped by various categories.

5.2.1 Framework Improvements

In this section we list some of the features that are needed, and some experiments that may (or may not) improve the framework itself.

5.2.1.1 Algorithmic Improvements

Some of the experiments that can be tried with the genetic algorithm are:

Solution Preservation Currently, the algorithm only maintains the solution as the evolving population. Therefore, the final solution can only be as large as the population size. Therefore, some points that are actually non-dominated may be lost in favour of others that, for example, may be improving the diversity of the overall solution. A useful improvement would be to save a separate set, unrestricted by size, of all the non-dominated points.

Real Variable Encoding It may be interesting to see if the convergence or quality of the solutions improve if the variables are encoded as real variables instead of binary as they are in this work. Real encoding per variable with an appropriate crossover operator can stop crossover from disrupting the contents of the variables, thereby preserving more information. This may nor may not be useful, but is a simple and interesting experiment.

5.2.1.2 Parallelization Changes

The current framework uses Remote Procedure Calls for parallelism. However, newer versions of Linux do not support user-mode RPC. Therefore, the parallelization framework needs to be changed to a more supported library. In order of desirability, this new framework can be – MPI (least effort), XML-RPC, or Java RMI. The last two also contribute in making the system more modular.

5.2.1.3 Increasing Modularity

Although much attempt has been made in the current work towards modularity, the following additional changes may be incorporated for further improvement:

Object Oriented Approach From the experience of the present work, we believe converting the framework to an object oriented approach will go a long way in making the system more modular. As our final goal, we would like to support many different simulators. This is made much easier by the template based object oriented approach – users may submit their own fitness function object, which can be more modularly (and dynamically) loaded on to the framework.

XML support An alternative to the above approach is support for standard XML interfaces to the framework – users may submit their own simulators following the XML template.

Both of these approaches also make it possible for the users to run their own (possibly propriety) simulators on their own systems, and use our genetic framework remotely.

5.2.1.4 New Analysis Metrics

We found that metrics for comparing Pareto fronts were not very well studied, with each author giving their own metric for comparison. A single good metric for representing the quality of a Pareto front, in the absence of the ideal front for reference, is needed.

5.2.2 Accuracy and Granularity Improvements

The results given in this work are limited by the accuracy of the simulators we use, which were a legacy from previous work in the project. Future work in the project may improve the accuracy/granularity of the results by the following means:

5.2.2.1 Better Cycle Accurate Simulation

We found many issues with the XTREM simulator:

- Changing frequency is not supported
- Memory leaks – running XTREM through a memory checker tool shows many memory leak problems
- Area analysis is not supported thereby necessitating the use of another simulator. Simulation of core-area would be appreciated

Therefore, work has already begun to use the Xeemu[37] simulator, which is an improvement over XTREM.

5.2.2.2 Support for Synthesis

Although we model a lot of the simulator features, we can get more accurate results if we also support synthesis. Lisatek [40] is a tool developed by CoWare that may help in describing modular components that can be encoded into the design space explorer, at the same time providing support for synthesis.

5.2.3 Support for Additional Features

5.2.3.1 Variable Frequency

As mentioned before, XTREM does not support variable frequencies. In subsequent explorations, we would also like to include frequency as part of the exploration.

5.2.3.2 Instruction Sets

We do not have support for changing instruction sets in this work. Lisatek, one of the new simulators under consideration, supports changing instruction sets. However, modeling instruction sets such that they can be changed by an automated design space explorer may make for challenging and interesting future work.

5.2.3.3 Reliability

Reliability is an important requirement of the SiMS processor. We would like to have reliability as one of the design space exploration objectives, so that we make design decisions based on optimal tradeoffs between reliability and power, performance and area. However, there is no easy method to model reliability such that an automated framework can selectively introduce various grades of fault tolerance.

5.2.3.4 Processor Sleep and Performance states

Many embedded processors have one or more 'sleep states'. During time of inactivity, the processor switches off some components in order to conserve power. This comes at a loss of performance as it usually takes some time to restore the functionality when required. Similarly, some processors also support voltage/frequency scaling in order to tradeoff power and performance. These states may become very important in a resource constrained application such as implants. Therefore, it would be interesting to model these for exploration.

Bibliography

[1] *Cell Relay Retreat: CRC-32 Calculation, Test Cases and HEC Tutorial*, `http://cell.onecall.net/cell-relay/publications/software/`.

[2] Giuseppe Ascia, Vincenzo Catania, and Maurizio Palesi, *Parameterised system design based on genetic algorithms*, CODES '01: Proceedings of the ninth international symposium on Hardware/software codesign (New York, NY, USA), ACM, 2001, pp. 177–182.

[3] K. Au-Yeung, C.R. Johnson, and P.D. Wolf, *A novel implantable cardiac telemetry system for studying atrial fibrillation*, Physiological Measurement **25** (2004), 1223.

[4] T. Austin, E. Larson, and D. Ernst, *SimpleScalar: An infrastructure for computer system modeling*, Computer (2002), 59–67.

[5] T. Back, *Optimal mutation rates in genetic search*, Proceedings of the Fifth International Conference on Genetic Algorithms, 1993, pp. 2–8.

[6] HP Benson and E. Sun, *Outcome space partition of the weight set in multiobjective linear programming*, Journal of Optimization Theory and Applications **105** (2000), no. 1, 17–36.

[7] R. Braden, D. Borman, and C. Partridge, *Computing the internet checksum*, SIGCOMM Comput. Commun. Rev. **19** (1989), no. 2, 86–94.

[8] E. Cantú-Paz, *A survey of parallel genetic algorithms*, Calculateurs Paralleles Reseaux et Systemes Repartis **10** (1998), no. 2, 141–171.

[9] C.B. Cho, W. Zhang, and T. Li, *Informed Microarchitecture Design Space Exploration Using Workload Dynamics*, Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, IEEE Computer Society, 2007, pp. 274–285.

[10] C.A.C. Coello, G.B. Lamont, and D.A. Van Veldhuizen, *Evolutionary algorithms for solving multi-objective problems*, Springer-Verlag New York Inc, 2007.

[11] G. Contreras, M. Martonosi, J. Peng, R. Ju, and G.Y. Lueh, *XTREM: a power simulator for the Intel XScale® core*, Language, Compiler and Tool Support for Embedded Systems **39** (2004), no. 7, 115–125.

[12] ARM corporation website, *http://www.arm.com/*, March 2010.

[13] P.S. Cross, R. Kunnemeyer, C.R. Bunt, D.A. Carnegie, and M.J. Rathbone, *Control, communication and monitoring of intravaginal drug delivery in dairy cows*, International Journal of Pharmaceuticals, vol. 282, 10 September 2004, pp. 35–44.

[14] I. Das and J.E. Dennis, *A closer look at drawbacks of minimizing weighted sums of objectives for Pareto set generation in multicriteria optimization problems*, Structural and Multidisciplinary Optimization **14** (1997), no. 1, 63–69.

[15] D. Dave, C. Strydis, and G. N. Gaydadjiev, *A Multidimensional, Design-Space Exploration Framework for Biomedical-Implant Processors*, Submitted to: Proceedings of the 21th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP'10), July 7-9 2010.

[16] Dhara Dave, Christos Strydis, and Georgi Gaydadjiev, *A multi-dimensional design space exploration framework for biomedical-implant processors*, pending publication.

[17] N.E. de Vries, *Lossless Data-Compression Kit, LDS v1.3*, `http://www.nicodevries.com/nico/lds13.zip`.

[18] K. Deb, *Multi-Objective Optimization Using Evolutionary Algorithms*, John Wiley & Sons, LTD, 2001.

[19] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan, *A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II*, Parallel Problem Solving from Nature PPSN VI, Springer, 2000, pp. 849–858.

[20] Merriam-Webster Online Dictionary, *Optimization*, `http://www.merriam-webster.com/dictionary/optimization`, September 2009.

[21] M. Dorigo, G.D. Caro, and L.M. Gambardella, *Ant algorithms for discrete optimization*, Artificial life **5** (1999), no. 2, 137–172.

[22] T. Eggers, C. Marschner, U. Marschner, B. Clasbrummel, R. Laur, and J. Binder, *Advanced hybrid integrated low-power telemetric pressure monitoring system for biomedical applications*, IEEE Proceedings of Microelectromechanical Systems (MEMS) (Miyuzaki, Japan), 2000, pp. 329–334.

[23] ______, *Advanced hybrid integrated low-power telemetric pressure monitoringsystem for biomedical applications*, MEMS'00, 2000, pp. 329–334.

[24] TU Delft Computer Engineering, *The SiMS project*, `http://ce.et.tudelft.nl/SiMS/`, December 2009.

[25] Stijn Eyerman, Lieven Eeckhout, and Koen De Bosschere, *Efficient design space exploration of high performance embedded out-of-order processors*, DATE '06: Proceedings of the conference on Design, automation and test in Europe (3001 Leuven, Belgium, Belgium), European Design and Automation Association, 2006, pp. 351–356.

[26] C.M. Fonseca, P.J. Fleming, et al., *Genetic algorithms for multiobjective optimization: Formulation, discussion and generalization*, Proceedings of the fifth international conference on genetic algorithms, Citeseer, 1993, pp. 416–423.

[27] F. Glover, E. Taillard, and E. Taillard, *A user's guide to tabu search*, Annals of operations research **41** (1993), no. 1, 1–28.

[28] D.E. Goldberg and J.H. Holland, *Genetic algorithms and machine learning*, Machine Learning **3** (1988), no. 2, 95–99.

[29] V.S. Gordon and D. Whitley, *Serial and parallel genetic algorithms as function optimizers*, Proceedings of the Fifth International Conference on Genetic Algorithms, 1993, pp. 177–183.

[30] JJ Grefenstette, *Optimization of control parameters for genetic algorithms*, IEEE Transactions on Systems, Man and Cybernetics **16** (1986), no. 1, 122–128.

[31] Matthias Gries, *Methods for evaluating and covering the design space during early design development*, Integr. VLSI J. **38** (2004), no. 2, 131–183.

[32] T.D. Gwiazda, *Crossover for single-objective numerical optimization problems*, Tomasz Gwiazda, 2006.

[33] YY Haimes, LS Lasdon, and DA Wismer, *On a bicriterion formulation of the problems of integrated system identification and system optimization*, IEEE Transactions on Systems, Man, and Cybernetics **1** (1971), no. 3, 296–297.

[34] O. Hasançebi and F. Erbatur, *Evaluation of crossover techniques in genetic algorithm based optimum structural design*, Computers and structures **78** (2000), no. 1-3, 435–448.

[35] GJ Hekstra, GD La Hei, P. Bingley, and FW Sijstermans, *TriMedia CPU64 design space exploration*, iccd, Published by the IEEE Computer Society, 1999, p. 599.

[36] J.L. Hennessy, D.A. Patterson, D. Goldberg, and K. Asanovic, *Computer architecture: a quantitative approach*, Morgan Kaufmann, 2003.

[37] Z. Herczeg, Á. Kiss, D. Schmidt, N. Wehn, and T. Gyimothy, *XEEMU: An improved XScale power simulator*, Lecture Notes in Computer Science **4644** (2007), 300.

[38] K. Hille, J. Draeger, T. Eggers, and P. Stegmaier, *[Technical construction, calibration and results with a new intraocular pressure sensor with telemetric transmission] [Article in German]*, Klinische Monatsblatter fur Augenheilkunde, vol. 218, May 2001, pp. 376–380.

[39] J.H. Holland, *Genetic algorithms*, Scientific American **267** (1992), no. 1, 66–72.

[40] CoWare Inc, *LISATek*, Retrieved from http://www.coware.com.

[41] A. Iorio and X. Li, *Rotationally Invariant Crossover Operators in Evolutionary Multi-objective Optimization*, Lecture Notes in Computer Science **4247** (2006), 310.

[42] T. Jansen and I. Wegener, *On the choice of the mutation probability for the (1+ 1) EA*, Lecture notes in computer science (2000), 89–98.

[43] Vinod Kathail, Shail Aditya, Robert Schreiber, B. Ramakrishna Rau, Darren C. Cronquist, and Mukund Sivaraman, *Pico: Automatically designing custom computers*, Computer **35** (2002), 39–47.

[44] I.Y. Kim and O.L. de Weck, *Adaptive weighted sum method for multiobjective optimization: a new method for Pareto front generation*, Structural and Multidisciplinary Optimization **31** (2006), no. 2, 105–116.

[45] S. Kirkpatrick, *Optimization by simulated annealing: Quantitative studies*, Journal of Statistical Physics **34** (1984), no. 5, 975–986.

[46] A. Konak, D.W. Coit, and A.E. Smith, *Multi-objective optimization using genetic algorithms: A tutorial*, Reliability Engineering and System Safety **91** (2006), no. 9, 992–1007.

[47] HP Labs, *Cacti*, `http://www.hpl.hp.com/research/cacti/`, September 2009.

[48] Y.W. Law, J. Dourmen, and P. Hartel, *Survey and benchmark of block ciphers for wireless sensor networks*, ACM Transactions on Sensor Networks **2** (2006), 65–93.

[49] C.K. Liang, J.J.J. Chen, C.L. Chung, C.L. Cheng, and C.C. Wang, *An implantable bi-directional wireless transmission system for transcutaneous biological signal recording*, Physiological Measurement **26** (2005), 83–97.

[50] F. Mattern and P. Sturm, *From distributed systems to ubiquitous computing-the state of the art, trends, and prospects of future networked systems*, Kommunikation in Verteilten Systemen (KiVS) 2003 (K. Irmscher and K.-P. Fähnrich, eds.), 2003, pp. 3–25.

[51] G. Mavrotas, *Effective implementation of the [epsilon]-constraint method in multi-objective mathematical programming problems*, Applied Mathematics and Computation **213** (2009), no. 2, 455 – 465.

[52] Sun Microsystems, *Onc+ developers guide*, Sun Microsystems, Inc, May 2002.

[53] S. Mohanty, V. K. Prasanna, S. Neema, and J. Davis, *Rapid design space exploration of heterogeneous embedded systems using symbolic search and multi-granular simulation*, LCTES/SCOPES '02: Proceedings of the joint conference on Languages, compilers and tools for embedded systems (New York, NY, USA), ACM, 2002, pp. 18–27.

[54] H. Muhlenbein, *How genetic algorithms really work: I. mutation and hillclimbing*, Parallel problem solving from nature **2** (1992), 15–25.

[55] M.F.X.J. Oberhumer, *LZO v2.0.2*, `http://www.oberhumer.com/opensource/lzo/`.

[56] H. Ohta and M. Matsui, *A Description of the MISTY1 Encryption Algorithm*, United States, 2000.

[57] A.D. Pimentel, L.O. Hertzberger, P. Lieverse, P. van der Wolf, and F. Deprettere, *Exploring embedded-systems architectures with Artemis*, Computer (2001), 57–63.

[58] S. Pourmehdi, P. Strojnik, P.H. Peckham, J.R. Buckett, and B. Smith, *A custom-designed chip to control an implantable stimulator and telemetry system for control of paralyzed muscles*, Proceedings of the 6th Vienna International Workshop on Functional Electrical Stimulation (Vienna, Austria), 22?24 September 1998.

[59] ______, *A custom-designed chip to control an implantable stimulator and telemetry system for control of paralyzed muscles*, Artificial Organs, vol. 23, May 1999, pp. 396–398.

[60] WordNet 3.0 © Princeton University, 2006, *Mutation*, `wordnetweb.princeton.edu/perl/webwn/mutation`, September 2009.

[61] SiMS project, *http://sims.et.tudelft.nl ¿¿ downloads*, March 2010.

[62] K. Qureshi and H. Rashid, *A performance evaluation of RPC, Java RMI, MPI and PVM*, Malaysian Journal of Computer Science **18** (2005), no. 2, 38–44.

[63] DL Rollins, CR Killingsworth, GP Walcott, RK Justice, RE Ideker, and WM Smith, *A telemetry system for the study of spontaneous cardiac arrhythmias*, IEEE Transactions on Biomedical Engineering **47** (2000), no. 7, 887–892.

[64] S.J. Russell, P. Norvig, J.F. Canny, J. Malik, and D.D. Edwards, *Artificial intelligence: a modern approach*, Prentice hall Englewood Cliffs, NJ, 1995.

[65] J. R. Schott, *Fault tolerant design using single and multi-criteria genetic algorithms*, Master's thesis, Department of Aeronautics and Astronautics, Massachusetts Institute of Technology, 1995.

[66] P. Shivakumar and N.P. Jouppi., *Cacti 3.0: An integrated cache timing, power, and area model.*, Tech. Report TN-2001/2, Compaq Western Research Laboratory, August 2001.

[67] B. Smith, Z. Tang, M.W. Johnson, S. Pourmehdi, M.M. Gazdik, J.R. Buckett, and P.H. Peckham, *An externally powered, multichannel, implantable stimulator-telemeter for control of paralyzed muscle*, IEEE Transactions on Biomedical Engineering, vol. 45, 1998, pp. 463–475.

[68] F.J. Solis and R. JB, *Minimization by random search techniques*, Mathematics of Operations Research **6** (1981), no. 1, 19–30.

[69] W.M. Spears, *Adapting crossover in a genetic algorithm*, Naval Research Laboratory AI Center Report AIC-92 **25** (1992).

[70] Y.N. Srikant and Priti Shankar (eds.), *The compiler design handbook: Optimizations and machine code generation*, second ed., ch. 3.3, CRC Press, 2008.

[71] C. Strydis, *Implantable microelectronic devices: A comprehensive review*, Master's thesis, Computer Engineering, Delft University of Technology, 2005.

[72] ______, *Suitable cache organizations for a novel biomedical implant processor*, IEEE International Conference on Computer Design, 2008. ICCD 2008, 2008, p. 591598.

[73] C. Strydis et al., *Implantable microelectronic devices: A comprehensive review*, CE-TR-2006-01, Computer Engineering, TU Delft, Dec. 2006.

[74] C. Strydis and G.N. Gaydadjiev, *The case for a generic implant processor*, Annual International Conference of the IEEE Engineering in Medicine and Biology Society. IEEE Engineering in Medicine and Biology Society. Conference, vol. 1, 2008, p. 3186.

[75] ______, *Profiling of lossless-compression algorithms for a novel biomedical-implant architecture*, Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis, 2008, p. 109114.

[76] ______, *Profiling of symmetric-encryption algorithms for a novel biomedical-implant architecture*, Proceedings of the 2008 conference on Computing frontiers, 2008, p. 231240.

[77] C. Strydis and G.N. Gaydadjiev, *Evaluating Various Branch-Prediction Schemes for Biomedical-Implant Processors*, Proceedings of the 2009 20th IEEE International Conference on Application-specific Systems, Architectures and Processors-Volume 00, IEEE Computer Society, 2009, pp. 169–176.

[78] C. Strydis, C. Kachris, and G.N. Gaydadjiev, *ImpBench-A novel benchmark suite for biomedical, microelectronic implants*, International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS08), p. 2124.

[79] Christos Strydis, Dhara Dave, and Georgi Gaydadjiev, *Impbench revisited: An extended characterization of implant-processor benchmarks*, pending publication.

[80] G. Sywerda, *Uniform crossover in genetic algorithms*, Proceedings of the third international conference on Genetic algorithms table of contents, Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 1989, pp. 2–9.

[81] J. Scott J. Sztipanovits S. Asaad T. Bapty, S. Neema, *Model-integrated tools for the design of dynamically reconfigurable systems*, Tech. report, ISIS, Vanderbilt University, 2000.

[82] D. Thain, T. Tannenbaum, and M. Livny, *Distributed computing in practice: The Condor experience*, Concurrency and Computation Practice and Experience **17** (2005), no. 2-4, 323–356.

[83] L. Thiele, S. Chakraborty, M. Gries, and S. K
"unzli, *Design space exploration of network processor architectures*, Network Processor Design: Issues and Practices **1** (2002), 55–89.

[84] P. Valdastri, A. Menciassi, A. Arena, C. Caccamo, and P. Dario, *An implantable telemetry platform system for in vivo monitoring of physiological parameters*, IEEE Transactions on Information Technology in Biomedicine, vol. 8, September 2004, pp. 271–278.

[85] D. A. V. Veldhuizen and G. B. Lamont, *Evolutionary computation and convergence to a pareto front*, Genetic Programming 1998: Proceedings of the Third Annual Conference (University of Wisconsin, Madison, WI, USA) (John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba, and Rick Riolo, eds.), Morgan Kaufmann, 22-25 July 1998.

[86] D. Whitley, *A genetic algorithm tutorial*, Statistics and computing **4** (1994), no. 2, 65–85.

[87] P. Wouters, M. De Cooman, D. Lapadatu, and R. Puers, *A low power multi-sensor interface for injectable microprocessor-based animal monitoring system*, Sensors and Actuators A: Physical, vol. 41-42, 1994, pp. 198–206.

[88] Y. Xie, G.H. Loh, B. Black, and K. Bernstein, *Design space exploration for 3D architectures*, ACM Journal on Emerging Technologies in Computing Systems (JETC) **2** (2006), no. 2, 65–103.

[89] E. Zitzler, K. Deb, and L. Thiele, *Comparison of multiobjective evolutionary algorithms: Empirical results*, Evolutionary computation **8** (2000), no. 2, 173–195.

[90] E. Zitzler, M. Laumanns, L. Thiele, et al., *SPEA2: Improving the strength Pareto evolutionary algorithm*, EUROGEN, Citeseer, 2001, pp. 95–100.

Appendices

A Papers

A.1 ImpEDE: A Multidimensional Design-Space Exploration Framework for Biomedical-Implant Processors

The following paper has been accepted at ASAP 2010 - The 21st IEEE International Conference on Application-specific Systems, Architectures and Processors.

ImpEDE: A Multidimensional Design-Space Exploration Framework for Biomedical-Implant Processors

Dhara Dave, Christos Strydis, Georgi N. Gaydadjiev
Computer Engineering Lab, Delft University of Technology, P.O. Box 5031, 2600 GA Delft, The Netherlands
E-mail: C.Strydis@tudelft.nl, D.Dave@student.tudelft.nl, G.N.Gaydadjiev@tudelft.nl

***Abstract*—The demand for biomedical implants keeps increasing. However, most of the current implant design methodologies involve custom-ASIC design. The SiMS project aims to change this process and make implant design more modular, flexible, faster and extensible. The most recent work within the SiMS context provides ImpEDE, a framework based on a multiobjective genetic algorithm, for automatic exploration of the design space of implant processors. The framework provides the processor designer with a Pareto front through which informed decisions can be made about specific implant families after analyzing their particular tradeoffs and requirements. A highly efficient, parallelized version of the genetic algorithm is also used to evolve the front and has as its objectives the optimization of power, performance and area. In addition, we illustrate the extensibility of our framework by modifying it to include a case study of a synthetic implant application with hard realtime deadlines.**

***Keywords*-Design-space exploration; simulation; optimization; genetic algorithms; biomedical microelectronic implant; power consumption**

I. Introduction

Each company aims at organizing its design process around frameworks that allow for reuse of previously generated knowledge, leading to quicker *development* and shorter *time to market*. This need for reutilization has, surprisingly, not touched medical implants which – as shown in [1] – are more often than not, still developed *from scratch*. Yet, ballooning healthcare costs and increasing social debates on medical spending make this practice quite unacceptable. Besides, the faster new implants come to the market, the more patients benefit both in terms of quality of life and reduced cost.

From an engineering standpoint, medical implants constitute one highly resource-constrained class of embedded systems. For example, an implant must be small enough to fit into the implantation site. The peak power consumption must not be so high that the processor heats up and damages the surrounding tissue. Since frequent surgery to replace batteries would be detrimental to the quality of life of the patient, as well as very costly, the system must last long enough to provide its task without running out of power. Most implant applications perform real-time monitoring and/or control, therefore, the implant must be able to handle these without failing. Most importantly, since the application may be life-critical, the system must be extremely fault-tolerant and reliable.

A core component of an implant, its processor must also fulfill all the above criteria. Given these strict design limitations, currently existing, general-purpose processors are not ideal for implant applications. There is, instead, a need for new processors that are better suited for use in a wide range of implant applications. Accordingly, and as part of the ongoing SiMS project [2], effort has been put on developing a novel biomedical-implant processor. Optimal cache and branch-prediction subsystems for this processor have already been studied in [3], [4]. These studies have offered many design insights yet provide local and, by necessity, biased optimizations. It is, yet, well-known that the global optimization of multiple design objectives – such as performance, power and area – across *all* processor subsystems is a non-trivial task. One must explore all possible processor configurations, compute the corresponding design objectives and find the *Pareto-dominant* solutions to be included in the final trade-off set. Since, typically, the design parameters that affect a processor are numerous, computing the behavior of all their possible combinations is quite hard, if not impossible. For example, by considering 13 processor *design parameters* represented by 36 (binary) bits, if one were to simulate all combinations, one would need to evaluate $2^{36} = 68,719,476,736$ different processor configurations to identify the true *Pareto front* – an unrealistically high number. To make things worse, in this new field of implant processors there is no established set of processor characteristics that would allow meaningful limiting of the above number of potential configurations.

We therefore need a way to realistically approximate this ideal trade-off set without performing all possible simulations. Furthermore, even in an approximated scenario, thousands of configurations might still have to be evaluated and compared to get a good approximation of the true Pareto front. Hence, we need an automated method for doing so.

In this paper we present a novel framework intended for the systematic, rapid and adaptable design-space exploration (DSE) of processor architectures suitable for biomedical implants. The framework is termed ImpEDE (Implant-processor, Evolutionary, Design-space Explorer) and provides careful investigation of the processor design space through the use of a particular *genetic-algorithm* (GA) variant called NSGA-II, along with cycle-accurate simulations, considering realistic design constraints imposed by our prior knowledge of the field. This implementation featured a very long computational time and, therefore, a parallelized version of the algorithm has also been implemented and described in the current document. Concisely, the contributions of this work are:

- A first yet educated attempt towards the systematic, automated and accurate design of implant processors;
- A fine-tuned toolset that delivers optimized implant-processor configurations across multiple first-order (e.g. performance, power) and second-order (e.g. hard real-time deadlines) objectives; and
- A freely available parallelized version that can be expanded with additional design objectives and constraints and extended to applications classes other than implants.

The rest of the paper is organized as follows: section II briefly discusses other DSE toolsets available and explains the necessity for the current framework. Section III provides the overview and organization of ImpEDE while section IV discusses the performed customizations and selected parameters for making the framework ideal for architectural exploring of implant processors. Section V displays the validity of ImpEDE by presenting actual DSE results for our targeted implant processor and illustrates its flexibility by extending it with exploration under hard realtime constraints. Overall conclusions and future work are drawn in section VI.

II. Related work

A large number of design-space exploration tools have been presented in the past, targeting generic DSPs [5] to network processors [6] and, more recently, multicore processors [7], [8]. For an extensive overview of different approaches, the interested reader can refer to Matthias Gries [9].

One of the proposed methods is to employ Genetic Algorithms for design-space exploration purposes. Ascia et al. [10] have attempted to optimize processor subsystems yet the search-and-optimize algorithm they used is working only on a single objective at a time. Ghali and Hammani [11], on the other hand, have made use of a multiobjective setup, similar to ours, but address the optimization of Turbo decoders. Stijn et al. [12] have investigated various, automated, GA-based, single- and multi-objective DSE methods for custom processors, but focus on out-of-order flavors only.

To the best of our knowledge, none of the investigated tools was explicitly concerned with implantable systems. Yet, the field of biomedical, microelectronic implants is new and fast-progressing and, as discussed in the previous section, it calls for particular design constraints such as ultra-low power consumption, high fault-tolerance levels and tight execution deadlines.

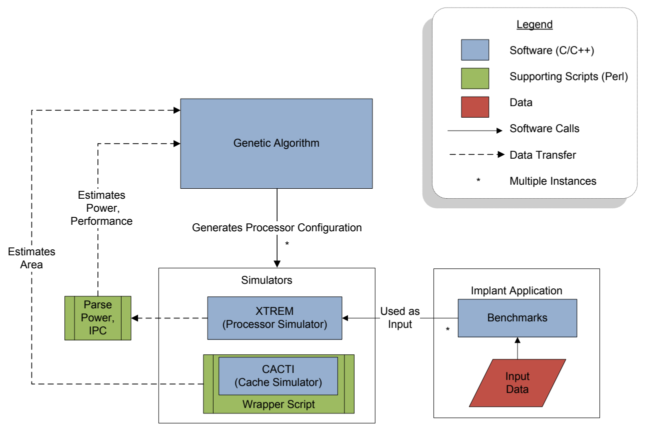


Figure 1. Framework organization.

What is needed is a fresh top-down approach to the field where implant applications are extensively profiled in a properly fine-tuned environment and the findings are used to drive an (automated, if possible) design-exploration effort for a suitable implant device. Setting up such an environment is a non-trivial problem as its specific parameters are either unknown or undisclosed, subject to tight proprietary controls. Building on our previous knowledge, with this work, we attempt to put together a multiobjective, DSE environment powered by suitable biomedical benchmarks and workloads. Also, we attempt to fine-tune this environment to the goal at hand by providing the first – in our knowledge – tool to offer bounded DSE. Last, we make this tool freely available for further improvement, expansion and dissemination of information.

III. Framework organization

Before introducing the DSE framework, we first have to identify the nature of the problem we attempt to solve: In designing our implant processor, we have formulated our problem as a multiobjective-minimization problem, with the objectives being processor *latency*, *average power consumption* and *area cost*. This is a set of first-order objectives typically optimized for in digital design. In the future and as the framework matures, we wish to add more objectives in our design effort such as *fault coverage* and more constraints such as *hard realtime deadlines*. In later sections, we will exemplify the latter by imposing a hard deadline on execution time (i.e. latency).

Since our objectives are minimizing area and power while maximizing performance, in order to convert our problem to a fully minimizing problem, we need to take the complement of performance as the objective encoded in our framework. We use IPC as the metric of performance. Therefore, we can simply use IPC*(-1) as the objective to be minimized. [1] For the rest, we use the metrics mm^2 and mW as the area and the power objectives, respectively.

With these objectives in mind, we have designed the multiobjective, DSE framework shown in Fig. 1. At first, the selected GA (NSGA-II) generates valid processor configurations (i.e. a set of parameters) – also known as "chromosomes" – that are fed to a cycle-accurate, performance and power simulator (XTREM) and to a cache-area simulator (CACTI). The processor simulator also accepts as inputs implant-related benchmarks and assorted datasets (ImpBench). Then, both simulators execute and their resulting performance, power and area figures are fed back into the waiting GA which uses them to evaluate the *optimality* of the currently simulated processor configuration. This process is repeated a number of times equal to the preset chromosome *population*, then a few best-performing chromosomes – based on their fitness results – are *selected*, processed and propagated to the next round of optimizations, also known as *generation*. With each successive generation, increasingly better chromosomes are found and promoted; that is, we are approaching the true Pareto front for our DSE problem. Figure 2 shows the progress of the front at various evolution stages for a particular run of our framework. Although paper space limitations do not allow for an extensive analysis of all components used[2], in the following subsections, we will describe in more detail the components of the framework as well as the choices made on the GA parameters such as the population size, the number of generations and the chromosome-selection policy used.

A. Genetic algorithm: NSGA-II

The classical single-objective optimization methods can be used to perform multiobjective optimization by reformulating the multiobjective optimization problem into a single-objective one. This can either be done by combining the objectives into a single aggregate objective [14] or by only considering one of the objectives and moving the rest to a constraint set [15]. When designing this framework, we first used a single-objective GA that employed the weighted-sums approach for finding the fitness of an individual. However, this was quickly abandoned as we could not logically assign the values of the weights since there was no rationalization for preferring one objective over another without more information about the problem domain. Besides, there was no way of knowing the absolute upper and lower limits of the three objectives (performance, power, area). Finally, such approaches suffer from being unable to find solutions to problems having non-convex fronts. Therefore, special algorithms were formulated for multi-objective problems [16].

[1]Note that cycles per instruction (CPI) could have also been used: since CPI is the inverse of IPC, it would give an identical relative ordering of processor configurations as IPC*(-1).

[2]An extensive analysis can be found in [13].

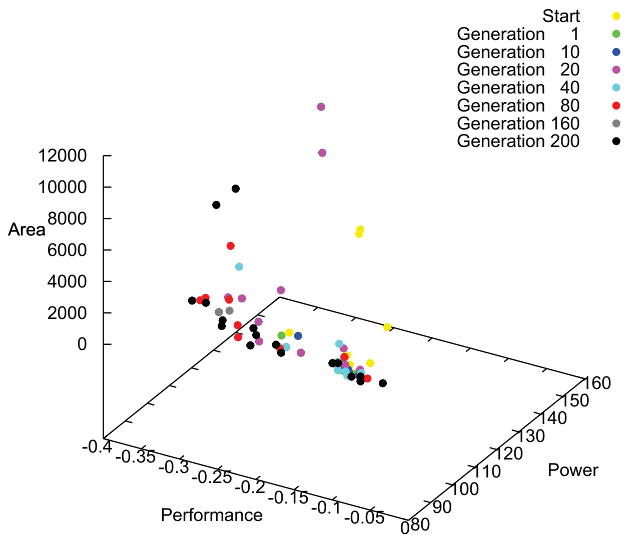


Figure 2. Framework-generated Pareto solutions (i.e. implant-processor instances).

NSGA [17] was one of the first GAs that evolve Pareto front solutions to multi-objective problems. NSGA-II [18] is the successor of NSGA that overcomes some of the limitations faced by the former. NSGA-II evolves Pareto fronts using an elitist approach and uses density and crowding distance metrics to ensure well spread out points along the front, at the same time having a lower computational complexity than its predecessor. It has therefore become in its own right widely accepted and used in diverse applications such as [19] and [20]. Due to its superiority over other algorithms, popularity, ease of use and availability, we use it as our algorithm of choice.

B. Processor & cache simulators

In the current version of our DSE framework, evaluation of the performance and power consumption of a given chromosome (i.e. processor configuration) has been based on the XTREM [21] **simulator**. XTREM simulator is a cycle-accurate, microarchitectural, power and performance simulator for the Intel XScale core [22]. It models the effective switching node capacitance of various functional units inside the core, following a similar modeling methodology to the one found in Wattch [23]. XTREM has been selected for its straight-forward functionality but mostly for its accuracy in performance and power modeling. It exhibits an average performance error of 6.5% and an average power error of 4% compared to real hardware.

XTREM allows monitoring of 14 different functional units of the Intel XScale core: Instruction Decoder (DEC), Branch-Target Buffer (BTB), Fill Buffer (FB), Write Buffer (WB), Pend Buffer (PB), Register File (REG), Instruction Cache (I$), Data Cache (D$), Arithmetic-Logic Unit (ALU),

Table I
ARCHITECTURAL DETAILS OF (MODIFIED) XTREM.

Feature	Value
ISA	32-bit ARMv5TE-compatible
Pipeline depth / width	7/8-stage pipeline / 32-bit
RF size	16 registers
Issue policy	in-order
Instruction window	single-instruction
I/D Cache L1 (separ.)	VAR size&assoc. (1-cc hit / 170-cc miss lat.)
BTB	VAR size, fully-assoc. / direct-mapped
Branch Predictor	VAR (4-cc mispred. lat.)
Ret. Address Stack	VAR size
I/D TLB (separ.)	1-entry / 1-entry
Write Buf. / Fill Buf.	2-entry / 2-entry
Mem. bus width	8-bit (1 mem. port)
INT/FP ALUs	1/1
Clock frequency	2 MHz
Implem. technology	0.18 μm @ 1.5 Volt

Shift Unit (SHF), Multiplier Accumulator (MAC), Internal Memory Bus (MEM), Memory Manager (MM) and Clock (CLK). We have extensively used XTREM in our previous studies on the implant processor and, in order to match our application field better, we have disabled many of XTREM's architectural parameters. In this case, however, we wish to allow for some degree of freedom in the processor design parameters so that the GA can explore a wider range of possible configurations. We have, thus, ended up with the (modified) XTREM characteristics summarized in Table I[3]. Performance/power figures have been checked and scaled properly with the changes. In the next section IV, we shall go through the selected simulator parameters and the way they have been encoded in the GA. It should be noted that flexible wrapper scripts have been used to provide the input to and capture the output of XTREM. As a result, the internal framework structure has been kept highly modular allowing for porting faster, more accurate or more powerful simulators in the future.

For evaluating the area cost of each chromosome, we have made the valid approximation that the subsystem dominating our envisioned implant processor is the cache (which holds also true for modern general-purpose processors). Furthermore, as can be seen from Table I, more adjustable parameters include some cache-like structure in them. Therefore, for quantifying each chromosome's area cost, we have used CACTI, a well-known, cache-area estimation tool. CACTI v3.2 has been primarily used since it is suitable for modeling simpler (older) cache-like structures (such as the BTB) and at an implementation technology identical to the one of the simulator (180 nm). However, the wrapper scripts we have created (Fig. 1) can also handle CACTI versions 4.1 and 6.0, if desired.

[3]Values denoted with 'VAR' indicate adjustable parameters by the GA.

Table II
IMPBENCH [24] BENCHMARKS. COLUMNS DENOTED WITH A (*) INDICATE AVERAGE VALUES FOR $1-KB$ INPUT WORKLOADS.

Benchmark type	Name	Size (KB)	#Instr.*	Sim. time* (sec)
Compression	miniLZO	16.30	199,163	3.07
	Finish	10.40	852,663	21.82
Encryption	MISTY1	18.80	1,268,465	16.65
	RC6	11.40	864,930	9.37
Data integrity	checksum	9.40	62,869	0.80
	CRC32	9.30	419,159	5.46
Real applications	motion	9.44	859,371	31.16
	DMU	19.50	36,808,268	37.25

C. Biomedical benchmarks & workloads

Eight suitable **benchmark applications** have been used for execution on XTREM for evaluating different chromosomes. They comprise the ImpBench benchmark suite [24] and consist of *lossless data compression* algorithms, *symmetric-key encryption* algorithms and *data-integrity* algorithms as well as representative code based on *real biomedical applications*. The benchmarks represent anticipated common tasks running on future implant processors and exhibit varied characteristics, as shown in Table II.

Typical biomedical readouts are often highly periodic signals (e.g. heart beat) or stable signals (e.g. blood temperature) which can, under specific circumstances, display gradual or abrupt changes in value (e.g. a sudden muscle contortion). We have collected and used various representative workloads capturing both stable as well as rapidly changing patterns. The original data has been provided from the BIOPAC (R) Student Lab PRO v3.7 Software. Paper-size limitations do not allow for an extensive description of the various workloads; a concise overview of the workload details is provided in Table III. Since reported literature [1] has revealed that typical implant data-memory sizes range from 1 KB to 10 KB, workloads of both sizes (1 KB and 10 KB) have been profiled.

Each chromosome represents a particular processor instance onto which each all benchmarks except *DMU* (which runs on a single, hard-coded input) are fed with each of the 7 workloads (of (1 KB or 10 KB)) and are executed. This accounts for a total of 50 benchmark runs for the 1 KB case and another 50 for the 10 KB case. As we shall see in subsection III-D, this is a substantial amount of (cycle-accurate) simulation time. In order to get practical results in our limited time frame and without loss of generality, for this paper we have selected and executed only the EMGII workload as it displays worst-case performance characteristics. In so doing, we have limited the number of runs to 8 per workload size. We, nonetheless, explore the design space for both sizes in order to investigate the effect workload size has on the Pareto-optimal solutions.

Table III
BIOMEDICAL WORKLOADS WITH DOUBLE-PRECISION (8-BYTE) DATA SAMPLES OF SIZES 1-KB AND 10-KB.

Dataset name	size (Bytes)	Samples (#)	duration (sec)
Electromyogram II (EMGII)	1147 / 9605	144 / 1201	0,288 / 2,402
Electroencephalogram (EEGI)	984 / 9616	123 / 1202	0,615 / 6,010
Electrocardiogram (ECGI)	912 / 9615	114 / 1202	0,114 / 1,202
Respiratory Cycle I (RCI)	1192 / 9520	149 / 1191	1,490 / 11,910
Pulmonary Function I (PFI)	1184 / 9240	148 / 1155	1,480 / 11,550
Skin Temperature (AEP)	1120 / 9736	140 / 1217	0,700 / 6,085
Blood Pressure (BP)	1128 / 9545	141 / 1198	0,282 / 2,396

D. Parallelization & optimization

As shown in Table II, evaluating a single processor configuration (i.e. one GA individual) with a single input (EMGII) and a single data size (1KB), across all 8 benchmarks takes on an average 125.58 seconds. Assuming the $10KB$ workloads run $10\times$ as slow as the $1KB$ workloads, except for the DMU benchmark and, considering optimization across all 7 Workload types, a full run of the GA with a population size of 20 and 200 generations will take approximately 343 days per result. We do not consider the GA run times in this calculation as the execution overhead of the parallelized GA was found to be negligible, contributing only 0.13 seconds per generation.

Since this run-time is quite prohibitive, we parallelized the evaluation stage of the GA so that different individuals are evaluated on idle CPUs of the group's laboratory machines. Hence, the speedup offered by our parallelized version is equal to $\sim P/\lceil P/N \rceil$, where P is the population size and N is the number of computers available. During the runtime of the GA, support scripts periodically search for and prepare free machines for the algorithm to use; these machines are used on the lowest priority in order to not disrupt regular usage. Therefore, this framework is expandable, modular and requires minimum dedicated resources.

It turns out that at any given time, we had around 20 machines available for computation. Therefore, the runtime has been effectively reduced to about 17.5 days for each run. As discussed before, we reduced this time further by only considering the worst-case input for each benchmark, EMGII, and we performed separate runs for each of the input sizes of 1 KB and 10 KB.

IV. FRAMEWORK FINE-TUNING

In the previous section, we went through the various building blocks of the DSE framework. Adjusting the framework to target implant processors requires, however, fine-tuning of the GA parameters and proper encoding (i.e. representation) of the chromosomes. In what follows, we go through such details that make our framework suitable for implant-processor design.

A. Chromosome encoding

Since GAs optimize the information encoded in the chromosomes, we needed to define a chromosomal representation for the processor parameters that the GA can work with. There are several chromosomal encoding strategies, the simplest being encoding each variable as a string of 1 and 0 bits. In this work we chose this encoding rather than *real encoding* [25] since the processor parameters encoded are integer values. Each chromosome is encoded as shown in Table IV. The table lists the processor variables chosen, their ranges as well as the encoding and decoding rules. The included variables are in agreement with the ones in Table I.

The processor *parameters* we chose to include in the search space depended both on what we wanted out of the processor and the capabilities and limitations of the simulators we had. As can be seen from Table IV, clock frequency has been encoded but was not used in this version of the GA; we found that running the simulator with different clock frequency values did not affect the results. For the purposes of this work, XTREM runs on a default clock frequency of 2 MHz, typical for implant processors.

The Write Buffer and the Fill Buffer included in XScale and shown in Table I help achieve better performance by hiding memory-access stalls when the core is running at a high clock frequency (e.g. 200 MHz). This is hardly the case for an implant processor; therefore, we did not encode the two buffers but rather fixed their sizes at the minimum supported by XTREM, i.e. exactly two entries. For similar reasons, Translation Lookaside Buffers (I/D-TLBs) have been excluded from the GA and fixed to a single-entry structure each.

As can be seen from Table IV, I-cache and D-cache structures have also been encoded in the chromosome. Although the simulator theoretically supports adjustable I-cache and D-cache latencies, we found that varying the I-cache latency above the default value of 1 often had the simulator hanging. Therefore, we only include D-cache latency as a variable, which we vary from 1 to 16 clock cycles. Note also that the above discussion pertains only to L1 caches. At a stage where embedded applications hardly have even an L1 cache, we think having L2 caches would be an overkill, and therefore exclude them from our exploration. This is also in agreement with the results in [3].

Except for the processor parameters, also their *ranges* have been defined: i) by the minimum and maximum allowed range that the simulators could support – so that we could capture as much of the design spectrum as possible – and ii) by the previous profiling studies within the SiMS Project [2].

B. Population size

The size of the population represents the maximum number of Pareto points the algorithm can find. If we pick too small a size for the population, we would have lesser tradeoffs available. On the other hand, the GA is $O(mN^2)$, where m is the number of objectives and N is the population size. Keeping these factors in mind, we

Table IV
PROCESSOR DESIGN PARAMETERS CONSIDERED IN THIS WORK, ENCODED AS 36 CHROMOSOMAL BITS.

Parameter		Encoded		Decoding	Remarks
Name (Ref)	**Range**	**Range**	**Bits**	**Formula**	
Core Clock Frequency ($Freq$)	[1...64]	[0...63]	6	$n+1$	Not used in current version
Branch Prediction ($Bpred$)	$Bimodal, Taken, notTaken$	[0...2]	2	-	$bit_0 = isBimod$, $bit_1 = isTaken$
Branch Target Buffer: Number of Sets (btb_nsets)	[32...128]	[0...5]	3	2^{n+5}	Only valid for $isBimod = TRUE$
Branch Target Buffer: Associativity (btb_assoc)	[1...32]	[0...5]	3	2^n	Only valid for $isBimod = TRUE$
Branch Prediction: Return Address Stack (RAS)	[0...8]	[0...4]	3	$floor(2^{n-1})$	-
L1 I/D-Cache: Number of Sets (I/D_nsets)	[1...8192]	[0...13]	4	2^n	-
L1 I/D-Cache: Block Size (I/D_bsize)	[8...32]	[0...2]	2	2^{n+3}	-
L1 I/D-Cache: Associativity (I/D_assoc)	[1...32]	[0...2]	3	2^n	-
L1 I/D-Cache: Replacement Policy (I/D_repl)	f, r, l	[0...2]	2	-	$bit_0 = isFifo$, $bit_1 = isRandom$
L1 D-Cache: Latency ($D\_latency$)	[1...16]	[0...4]	3	$floor(2^n)$	-
L1 I-Cache: Latency ($I\_latency$)	[1...16]	[0...4]	3	$floor(2^n)$	Not used in current version

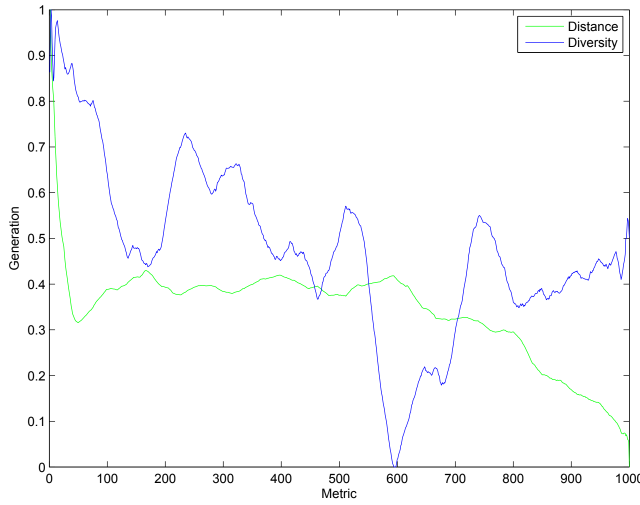


Figure 3. Smoothed distance and diversity metrics over 1000 generations (Benchmark: checksum)

chose a population size of 20, which also coincided with the number of machines we expected to be free at any given time. Therefore, the entire population could be evaluated at once in parallel.

C. Number of Generations

The number of generations represent the time the GA is allowed to reach the Pareto front. We observed that the GA converges rather rapidly to a front, then tries to increase its spread in the subsequent generations. After this, the GA reaches a sort of 'stable state' where it is unable to improve the spread without degrading the distance from the front, and vice versa. We can limit the number of generations at this phase, in order to reduce computation time. We use a reduced problem set - that of only optimizing the *checksum* benchmark, and run this for a large number of generations (1000) in order to approximate the number of generations needed, then use this as a guide to selecting the number of generations for the full problem.

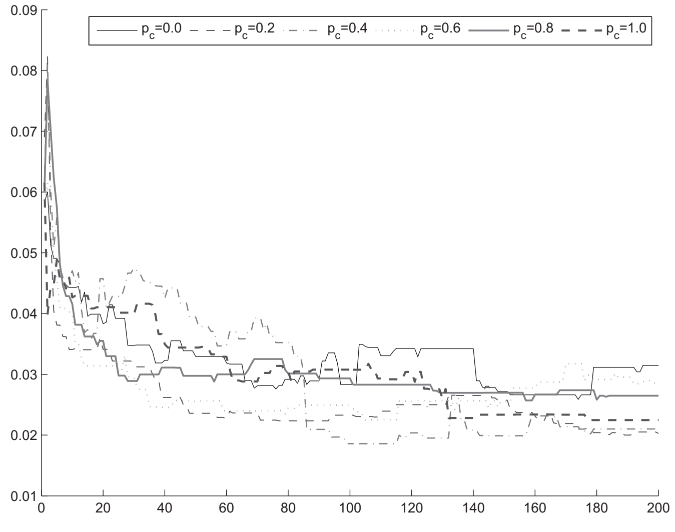
We use Deb's metric (Δ) [16] to quantify the diversity (spread) of the solutions and Veldhuizen's Generational Distance (GD) [26] metric to quantify the distance of the solution front from the 'true' Pareto front[4]. We found both metrics to be very noisy, specially the diversity metric. Therefore, to make them easier to compare, we smooth the data using a moving average with a span of 20 generations. Figure 3 shows the resulting metrics. We observer that GD declines rapidly until about the 100th generation, after which it fluctuates around GD=0.4 for a long time until finally dropping to zero around generation 600. The rapid decline is of course due to the solution fronts converging towards the reference. After generation 51, it can be seen that the general trend is that as spread decreases, distance increases and vice versa. The behavior from generations 100-593 are also expected - as the algorithm searches for new solutions, the distance fluctuates. A minor rise in both GD and spread may not mean that the front is actually further away than the one previously – since we have finite points in the reference front, points that are the same distance from the *actual* front may give slightly varying distance values from the reference front. The behavior at the tail end reflects the fact that the reference solution is a combination of several solutions, and therefore also contains the final solution of the run in question. The apparent rapid convergence seen therefore, is in fact the algorithm moving towards its own final solution – a foregone conclusion. Therefore, we do not consider this region in our analysis.

Since after generation 100, the algorithm seems to oscillate between improving spread and distance at the cost of the other; without loss of precision in both quantities at the same time, we can stop the algorithm around generation 100. Since GAs are random by nature, in order to be sure of getting good results, we run every subsequent experiment for twice this time, i.e. – 200 generations. This also compensates for the smoothing we performed.

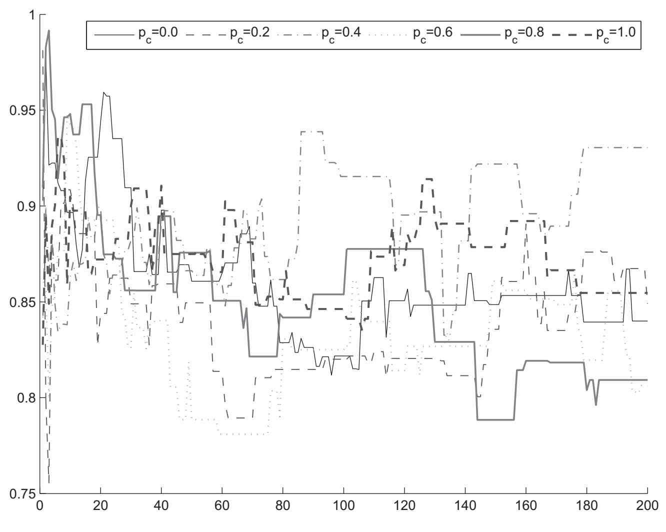
D. Mutation

There are a number of ways to select the mutation probability, including complicated adaptive mutation strategies

[4]Since we do not know the true front (by problem definition itself), we approximate it by computing a combined front consisting of mutually non-dominating points from the results of 10 separate runs of the algorithm. We call this the 'reference front'.



(a) Distance metric



(b) Diversity metric

Figure 4. Distance and Diversity metrics for various crossover probabilities (P_c) over 200 generations (Benchmark:checksum)

[27]. However, the simplest and most recommended strategy is simply setting the mutation probability as $p_m = 1/n$ where n is the chromosome length [28]. This means that a single bit per chromosome is expected to change from the parent to the child population. Therefore, the child chromosome is likely to vary from the parent chromosome in exactly one attribute, that too by a hamming distance of one. We use this approach for setting the mutation probability in our implementation of the GA, i.e $p_m = 1/36$.

E. Crossover probability

Crossover allows chromosomes to exchange information that may lead to better chromosomes that combine the "good qualities" of the parent chromosomes. Used carefully, crossover can lead to much quicker evolution times, and is central to the idea of Genetic Algorithms examining more solutions in the more promising regions of the solution space [29]. Therefore, it is important to set a good *crossover probability* P_c, which determines the percentage of chromosomes that undergo recombination at each generation. As in the case of number of generations, we used a reduced problem set – running only *checksum* with the $1 - KB$ EMGII input for 200 generations with different crossover probabilities. Figure 4 shows the two metrics for the Pareto fronts resulting from each value of P_c [5]. Keeping in mind the discussion from Section IV-C, we see from the graphs that $P_c = 0.2$ and $P_c = 0.6$ seem to lead to the fastest convergence and best values for the two metrics over the course of the generations, and therefore $P_c = 0.2$ is chosen for subsequent runs.

[5]Note that in this case, these are the un-normalized, un-smoothed metrics. They appear less noisy due to the fact that the number of generations plotted is much lower than in Section IV-C

V. Selected results & validation

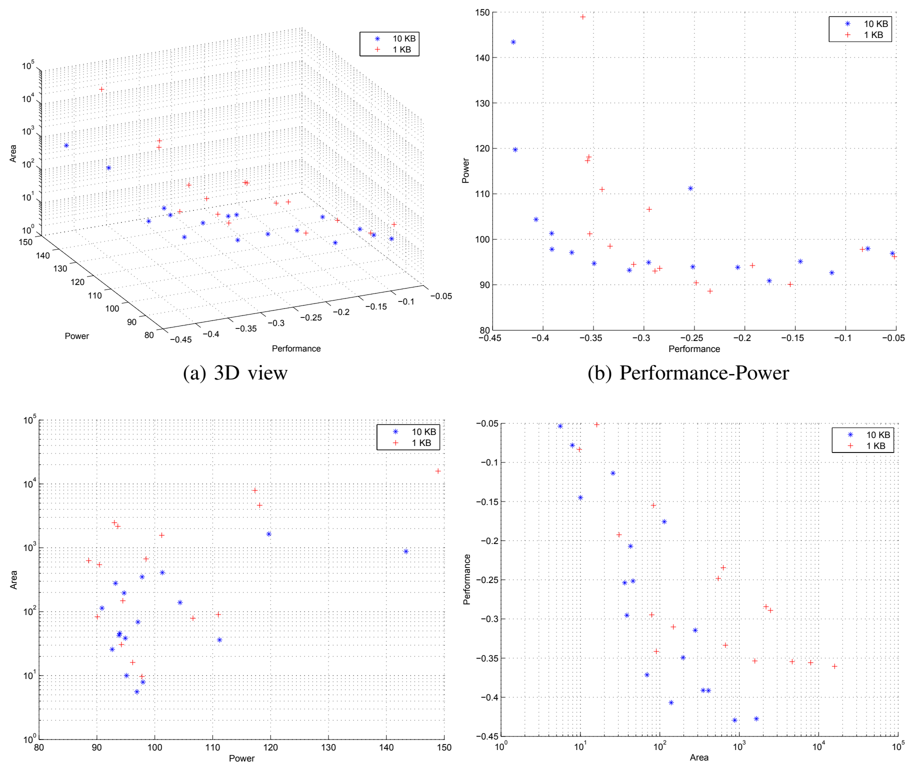
In this section, the correct functionality of the framework is demonstrated. Also, an expansion of the framework with a hard realtime deadline leading to constrained design is illustrated.

A. Implant-processor results

Having prepared and fine-tuned the framework to the best of our knowledge, we move on to testing it by running it with all the benchmarks, once for each of the two workload sizes. We call these the *baseline* results. Figure 5 shows the projections of the Pareto front evolved on the 3 Cartesian planes (performance, power, area)).

We readily see in both rows of results that the algorithm reaches the "front" by generation #40. After this, the points spread out and a wider Pareto front is found. We see from Fig. 5b and Fig. 5d that the $10 - KB$ workloads have a wider front w.r.t. performance. We anticipate this to be so because the bigger workload increases processor utilization by allowing the caches and branch predictor table to fill and, hence, minimizing processor stalls. This higher performance also leads to a corresponding increase in the power consumption as can be seen from the power axes in Fig. 5b and Fig. 5c.

On the contrary, we see slightly bigger area-utilization solutions for the smaller workload. Although measurements with more workload sizes are needed to draw safe conclusions, this area trend may again be due to "cold-start" effects; that is, due to the fact that when small workloads are processed, poor CPU utilization occurs since the cache structures do not have enough time to fill, pushing the GA towards larger structures in the hopes of minimizing cache stalls.



(a) 3D view (b) Performance-Power

(c) Power-Area (d) Performance-Area

Figure 5. Baseline DSE results for 1 KB and 10 KB workloads running on all benchmarks.

B. *Framework expansion*

As this was one of the first steps towards designing an implant processor, we wanted to make sure that the framework was expandable, in order to facilitate the addition of new domain specific information into the framework as it gets available. In order to test this property of the framework, we devised a synthetic implant application with a hard realtime deadline. Indeed, many implant applications have realtime requirements so formulating a synthetic problem with such a constraint does not fall far from practice. We modified the DMU and Motion benchmarks (called StressDMU and StressMotion respectively), to represent a single iteration of these (repetitive) applications. Further, this iteration was chosen to be the worst-case iteration (in terms of instruction-cycle count) for each of the two original benchmarks.

Combined with data-integrity checks, compression and encryption, the stress benchmarks represent an *atomic* action for an implant application – from data collecting, processing, to transferring – as exemplified in Fig. 6. In a real application, this atomic action must be finished, for example, before the next set of input arrives. Therefore, we *constrain* the total time required for this combined operation as a hard realtime deadline.

Out of the ImpBench set, we chose checksum, miniLZO and RC6 as the data-integrity, compression and encryption algorithms, respectively. We obtain the simulated execution time of the processor configuration under investigation from the simulator output. In case the deadline is violated, the processor configuration is deemed to be unacceptable. On the other hand, if the deadline is met, we calculate the objectives of the configuration by combining with the rest of the benchmarks in the test suite (including ”normal” DMU and Motion). Therefore, the fitness metric remains the same as the baseline case.

Figure 7 shows the Pareto front evolved with a deadline of 1 second, and also with a slightly relaxed deadline of 2 seconds. As expected, the stricter deadline encourages processor configurations that have a higher performance, but at the same time take slightly more power and area.

VI. Conclusions & future work

Although reliability is one of the major reasons for the need to design processors specifically for implants, the present work does not directly address reliability. Instead, it starts off with the idea that processor design can be looked at from a black-box design perspective and provides a flexible and modular framework for doing so. Given such a framework, adding reliability as one of the optimization objectives is the next logical step, left as future work. We would also like to expand the simulator models with more parameters such as (off-chip) memory, effectively allowing

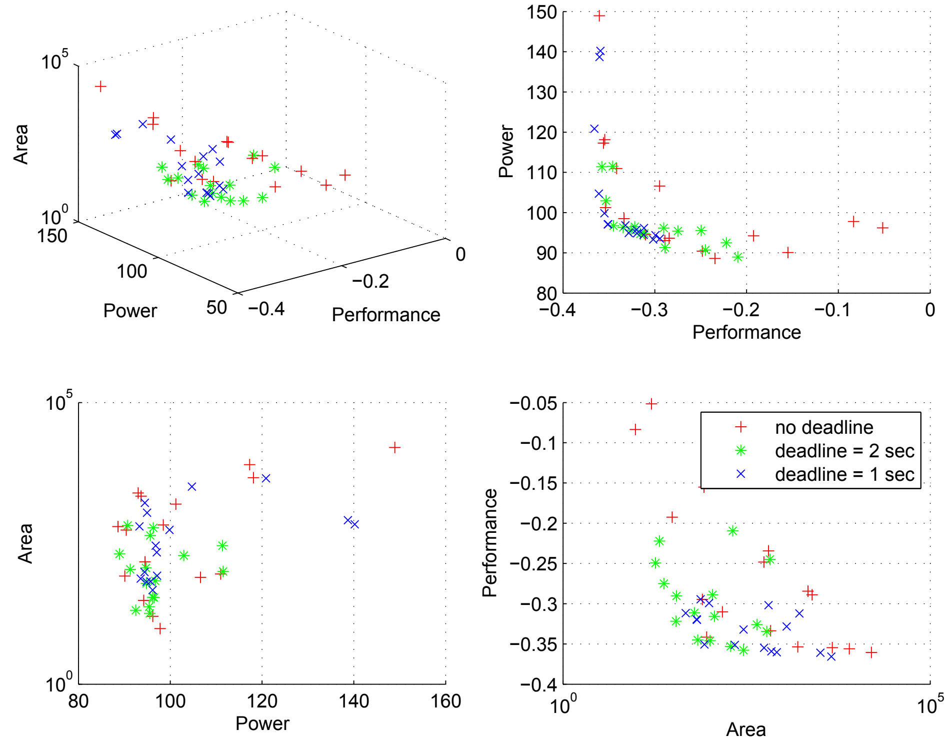


Figure 7. DSE results expanded with hard realtime deadlines of 2 *seconds* and 1 *second* for 10 *KB* workloads running on all benchmarks.

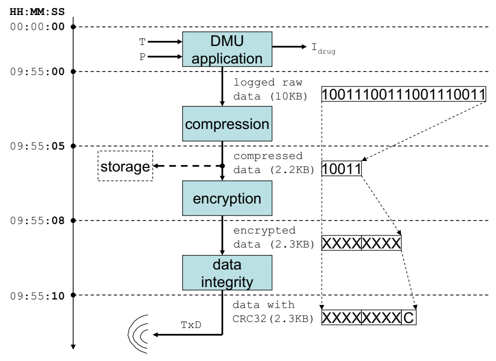


Figure 6. Conceptual block diagram of simulated implant application [30].

for System-on-Chip exploration. In order to overcome the aforementioned limitations of XTREM, we are considering the XEEMU [31] simulator as a candidate for the next phase of the framework.

Conclusively, in this paper we have developed ImpEDE, a novel, multiobjective, framework that provides high-level DSE of biomedical-implant processors, populated by suitable biomedical benchmarks and assorted workloads. ImpEDE organization is described in detail and its functionality is fine-tuned based on our previous experience (e.g. processor parameter values and ranges) and new findings (e.g. crossover probability, workload size). Restricted by its simulator components, the current framework version can deliver (near) Pareto-optimal processor solutions, co-optimized across performance, power consumption and area utilization. In view of potentially more optimization goals and benchmarks, we have paid attention to making the framework modular and expandable. Furthermore, we have provided a parallelized, versatile version of the framework which offers execution speedup roughly equal to the number of processor available, without dedicated hardware resources. Last, it has been our intension to make the proposed framework a freely accessible tool, available to everyone online for further studies and improvements.

VII. Acknowledgements

This work has been partially supported by the ICT Delft Research Centre (DRC-ICT) of the Delft University of Technology and by Google Inc.

References

[1] C. Strydis *et al.*, “Implantable microelectronic devices: A comprehensive review,” Computer Engineering, TU Delft, CE-TR-2006-01, Dec. 2006.

[2] “Smart implantable Medical Systems,” http://sims.et.tudelft.nl.

[3] C. Strydis and G. Gaydadjiev, “Suitable cache organizations for a novel biomedical-implant architecture,” in *International Conference of Computer Design (ICCD’08)*, Lake Tahoe, California, USA, 12-15 October 2008, pp. 591–598.

[4] C. Strydis and G. N. Gaydadjiev, "Evaluating Various Branch-Prediction Schemes for Biomedical-Implant Processors," in *Proceedings of the 20th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP'09)*, Boston, USA, July 2009, pp. 169–176.

[5] M. Lorenz, R. Leupers, P. Marwedel, T. Drager, and G. Fettweis, "Low-energy DSP code generation using a genetic algorithm," in *Proceedings of International Conference on Computer Design (ICCD) 2001, Austin, Texas*, 2001, pp. 431–437.

[6] L. Thiele, S. Chakraborty, M. Gries, and S. Künzli, "Design space exploration of network processor architectures," *Network Processor Design: Issues and Practices*, vol. 1, pp. 55–89, 2002.

[7] S. Mohanty, V. K. Prasanna, S. Neema, and J. Davis, "Rapid design space exploration of heterogeneous embedded systems using symbolic search and multi-granular simulation," in *LCTES/SCOPES '02: Proceedings of the joint conference on Languages, compilers and tools for embedded systems*. New York, NY, USA: ACM, 2002, pp. 18–27.

[8] A. Pimentel, L. Hertzberger, P. Lieverse, P. van der Wolf, and F. Deprettere, "Exploring embedded-systems architectures with Artemis," *Computer*, pp. 57–63, 2001.

[9] M. Gries, "Methods for evaluating and covering the design space during early design development," *Integr. VLSI J.*, vol. 38, no. 2, pp. 131–183, 2004.

[10] G. Ascia, V. Catania, and M. Palesi, "Parameterised system design based on genetic algorithms," in *CODES '01: Proceedings of the ninth international symposium on Hardware/software codesign*. New York, NY, USA: ACM, 2001, pp. 177–182.

[11] K. Ghali and O. Hammani, "Embedded Processor Characteristics Specification Through Multiobjective Evolutionary Algorithms," in *IEEE International Symposium on Industrial Electronics (ISIE'03)*, 2003, pp. 907–912.

[12] S. Eyerman, L. Eeckhout, and K. De Bosschere, "Efficient design space exploration of high performance embedded out-of-order processors," in *DATE '06: Proceedings of the conference on Design, automation and test in Europe*. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2006, pp. 351–356.

[13] D. Dave, "Automated implant-processor design: An evolutionary multiobjective exploration framework," Master's thesis, TU Delft, 2010.

[14] I. Kim and O. de Weck, "Adaptive weighted sum method for multiobjective optimization: a new method for Pareto front generation," *Structural and Multidisciplinary Optimization*, vol. 31, no. 2, pp. 105–116, 2006.

[15] G. Mavrotas, "Effective implementation of the [epsilon]-constraint method in multi-objective mathematical programming problems," *Applied Mathematics and Computation*, vol. 213, no. 2, pp. 455 – 465, 2009.

[16] K. Deb, *Multi-Objective Optimization Using Evolutionary Algorithms*. John Wiley & Sons, LTD, 2001.

[17] N. Srinivas and K. Deb, "Multiobjective optimization using nondominated sorting in genetic algorithms," *Evolutionary Computation*, vol. 2, pp. 221–248, 1994.

[18] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast elitist multi-objective genetic algorithm: Nsga-ii," *IEEE Transactions on Evolutionary Computation*, vol. 6, pp. 182–197, 2000.

[19] A. R. Price, I. I. Voutchkov, G. E. Pound, N. R. Edwards, T. M. Lenton, and S. J. Cox, "Multiobjective tuning of grid-enabled earth system models using a non-dominated sorting genetic algorithm (nsga-ii)," in *E-SCIENCE '06: Proceedings of the Second IEEE International Conference on e-Science and Grid Computing*. Washington, DC, USA: IEEE Computer Society, 2006, p. 117.

[20] E. G. Bekele and J. W. Nicklow, "Multi-objective automatic calibration of swat using nsga-ii," *Journal of Hydrology*, vol. 341, no. 3-4, pp. 165 – 176, 2007.

[21] G. Contreras *et al.*, "XTREM: A Power Simulator for the Intel XScale Core," in *LCTES'04*, 2004, pp. 115–125.

[22] *Intel XScale Microarchitecture for the PXA255 Processor: User's Manual*, Intel Corp., March 2003.

[23] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: a framework for architectural-level power analysis and optimizations," *SIGARCH Comput. Archit. News*, vol. 28, no. 2, pp. 83–94, 2000.

[24] C. Strydis, C. Kachris, and G. Gaydadjiev, "ImpBench - A novel benchmark suite for biomedical, microelectronic implants," in *To appear in International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS'08)*, Samos, Greece, 21-24 July 2008.

[25] A. H. Wright, "Genetic algorithms for real parameter optimization," in *Foundations of Genetic Algorithms*. Morgan Kaufmann, 1991, pp. 205–218.

[26] D. A. V. Veldhuizen and G. B. Lamont, "Evolutionary computation and convergence to a pareto front," in *Genetic Programming 1998: Proceedings of the Third Annual Conference*. University of Wisconsin, Madison, WI, USA: Morgan Kaufmann, 22-25 Jul. 1998.

[27] T. Jansen and I. Wegener, "On the choice of the mutation probability for the (1+ 1) EA," *Lecture notes in computer science*, pp. 89–98, 2000.

[28] T. Back, "Optimal mutation rates in genetic search," in *Proceedings of the Fifth International Conference on Genetic Algorithms*, 1993, pp. 2–8.

[29] J. Holland, "Genetic algorithms," *Scientific American*, vol. 267, no. 1, pp. 66–72, 1992.

[30] C. Strydis and G. Gaydadjiev, "The Case for a Generic Implant Processor," in *30th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC'08)*, August 2008, pp. 3186–3191.

[31] Z. Herczeg, Á. Kiss, D. Schmidt, N. Wehn, and T. Gyimothy, "XEEMU: An improved XScale power simulator," *Lecture Notes in Computer Science*, vol. 4644, p. 300, 2007.

A.2 ImpBench Revisited: An Extended Characterization of Implant-Processor Benchmarks

The following paper has been accepted at SAMOS X - International Conference on Embedded Computer Systems: Architectures, MOdeling and Simulation.

ImpBench Revisited: An Extended Characterization of Implant-Processor Benchmarks

Christos Strydis, Dhara Dave, Georgi N. Gaydadjiev
Computer Engineering Lab, Delft University of Technology,
P.O. Box 5031, 2600 GA Delft,The Netherlands
Phone:+31-(0)15-27-82326 E-mail: C.Strydis@tudelft.nl

Abstract—**Implants are nowadays transforming rapidly from rigid, custom-based devices with very narrow applications to highly constrained albeit multifunctional embedded systems. These systems contain cores able to execute software programs so as to allow for increased application versatility. In response to this trend, a new collection of benchmark programs for guiding the design of implant processors, ImpBench, has already been proposed and characterized. The current paper expands on this characterization study by employing a genetic-algorithm-based, design-space exploration framework. Through this framework, ImpBench components are evaluated in terms of their implications on implant-processor design. The benchmark suite is also expanded by introducing one new benchmark and two new stressmarks based on existing ImpBench benchmarks. The stressmarks are proposed for achieving further speedups in simulation times without polluting the processor-exploration process. Profiling results reveal that processor configurations generated by the stressmarks follow with good fidelity - except for some marked exceptions - ones generated by the aggregated ImpBench suite. Careful use of the stressmarks can seriously reduce simulation times up to x30, which is an impressive speedup and a good tradeoff between DSE speed and accuracy.**

Index Terms—**Implant, benchmark suite, stressmark, profiling, genetic algorithm, Pareto, kernel, power, energy.**

I. Introduction

In 1971, Abdel Omran in his - now - classic article on epidemiologic transition [1] has investigated the historical demography of populations, theorizing about three distinct periods of health transition: i) the era of pestilence and famine, ii) the era of receding pandemics, and iii) the era of man-made diseases. In the eve of the 21st century, Omran's theory has been verified as we are surely going through the third era of man-made, non-communicable diseases, better known as degenerative or chronic diseases. Characteristic of this transition has been a general pattern shift from dominating infectious diseases with very high mortality at younger ages, to dominating chronic diseases and injury, with lower overall mortality but peaks at older ages [2]. What makes chronic diseases an issue of interest is the presence, in developed countries, of a prevailing demographic trend of an ageing population.

Not all countries have undergone this transition in the same fashion, albeit developed countries are well into the transition path described by Omran. Yet, Omran's theory intended on stressing the fact that, when progressing from high to low mortality rates, all populations involve a shift in the causes of death and disease. True enough, developing and, primarily, developed countries are manifesting the effects predicted by the theorized shift such as low fertility rates, population growth, increased heart-disease and cancer rates. Given the two latter trends, the healthcare sectors strive for policy and institutional adaptation so as to safeguard the provision of healthcare services.

In addition to policy and institutional adaptations and updates, the healthcare sector has benefited from technological advancements coming from the disciplines of medicine as well as microelectronics. A technological niché of biomedical engineering that has particularly contributed is implantable microelectronic devices which act as a means to deal with monitoring and/or treatment of chronic patients. With the implantable pacemaker being the first and perhaps most well-known implant type - 299,705 such devices have been registered in Europe alone, by the year 2003 [3] - the road is being paved for numerous other applications [4]. An ever expanding implant list already contains monitors of body temperature, blood pressure or glucose concentration; also, intracardiac defibrillators (ICDs) and functional electrical stimulators for bladder control, for blurred eye cornea and so on.

Although biomedical implants have come a long way in terms of size and functionality, these days another shift in implant design is manifesting gradually. Nowadays, implant designers are taking advantage of the more widespread use of microprocessors while attempting shorter development times and more versatility out of traditionally hardwired implant devices. For these reasons they are opting for devices implementing their functionality based on software programs. That is, implants are transforming into fully-blown, embedded systems built around processor cores [5], [6], [7]. This shift is depicted in Fig. 1.

In this context, it becomes apparent that a day when definition of implant processors will be based on established ”implant benchmarks” is not far. In anticipation of this, a novel suite of implant-processor benchmarks, termed ImpBench, has

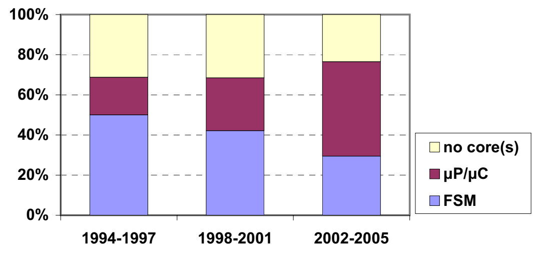


Fig. 1. Implant-core architecture trends over time [8].

been previously proposed and characterized [9][1]. Since its release, it has been put to good use in exploring efficient implant processors [10], [11] and in the work of others [12]. In this paper, we primarily attempt to expand the original characterization study with extensive, new results acquired through use of ImpEDE, a genetic-algorithm-(GA)-based, design-space exploration (DSE) framework [13]. Through using ImpEDE, we effectively investigate the *sensitivity* of different processor attributes (e.g. cache geometries) to the various benchmarks in ImpBench. We also update the suite with a new benchmark and two new, so-called "stressmarks", bringing ImpBench to version 1.1. Concisely, this paper contributes in:

- Providing a novel and sound methodology, based on GAs, of evaluating benchmark characteristics in terms of resulting processor configurations;
- Identifying a representative (subset of) benchmark(s) for substituting the whole suite in simulations, thus achieving radically shorter DSE times;
- Updating the ImpBench suite to version 1.1 by proposing: (a) a more sophisticated version of the DMU benchmark, and (b) two derived stressmarks for enabling shorter simulation times while biasing the exploration process insignificantly or, at least, predictably; and
- Reporting/amending errata of the original work and giving further clarifications, where needed.

The rest of the paper is organized as follows: section II gives an overview of previously proposed benchmark suites. In section III, the details of the components of ImpBench v1.1 are briefly reproduced. Section IV provides the details of our selected profiling testbed for benchmark evaluation. Section V presents, reflecting upon various metrics, the particular characteristics of each benchmark and its effect on the processor properties under investigation. The stressmarks are also introduced and evaluated. A summarizing discussion of the results is included in VI while overall conclusions and future work are discussed in section VII.

II. Related work

A large number of benchmark suites has already been proposed for various application areas, making covering the whole domain a far from trivial attempt. Instead, in this section we briefly discuss well-known and freely available benchmark suites, and mostly ones targeting the embedded domain, as is our implant case. The latest SPEC benchmark suite, CPU2006 [14], targets general-purpose computers by providing programs and data divided into separate integer and floating-point categories. MediaBench [15] is oriented towards increased-ILP, multimedia- and communications-oriented embedded systems. The Embedded Microprocessor Benchmark Consortium (EEMBC) [16] licenses "algorithms" and "applications" organized into benchmark suites targeting telecommunications, networking, digital entertainment, Java, automotive/industrial, consumer and office equipment products. It has also provided a suite capable of energy monitoring in the processor. EEMBC has also introduced a collection of benchmarks targeting multicore processors (MultiBench v1.0). MiBench [17] is another proposed collection of benchmarks aimed at the embedded-processor market. It features six distinct categories of benchmarks: automotive, industrial, control, consumer devices and telecommunications. MiBench bears many similarities with EEMBC, however it is composed of freely available source code. NetBench [18] has been introduced as a benchmark suite for network processors. It contains programs representing all levels of packet processing; from micro-level (close to the link layer), to IP-level (close to the routing layer) and to application-level programs. Network processors are also targeted by CommBench [19], focused on the telecommunications aspect. It contains eight, computationally intensive kernels, four oriented towards packet-header processing and four towards data-stream processing.

Compared with our own prior work [9], the new version of the ImpBench suite introduces a more detailed variation of the (originally described) *DMU* benchmark and two stressmarks, that is, two benchmarks based on *DMU* and *motion* and exhibiting worst-case execution (i.e. stress) behavior. A further novelty of the current work is the employment of a GA-based, DSE framework and analytic metrics in order to characterize the old and new ImpBench benchmarks. In effect, this paper extends the previous work in both terms of content and methodology. To the best of our knowledge, no benchmark suite has been published before to address the rising family of biomedical-implant processors. What is more, no characterization study has utilized GAs before to explore the benchmark properties and their implications on the targeted processor.

III. ImpBench v1.1 overview

In Table I are summarized all original and new ImpBench benchmarks. The significance and pertinence of these benchmarks to the implant context has been illustrated in [11]. The Table further reports binary sizes (built for ARM) and averaged, dynamic instruction/μop counts so as to give a measure of the benchmark complexity. We maintain the original grouping into four distinct categories: *lossless data compression*, *symmetric-key encryption*, *data-integrity* and *real applications*[2]. By including groups of different algorithms

[1]Available online: http://sims.et.tudelft.nl/, under "downloads".

[2]In the original ImpBench paper [9], real applications have also been called "synthetic benchmarks". However, this term is inaccurate and has, thus, been completely omitted from this work on. A suitable alternative to "real applications" could be the term "kernel".

TABLE I
IMPBENCH V1.1 BENCHMARKS.

benchmark	name	size (KB)	dyn. instr.* (average) (#)	dyn. μops* (average) (#)
Compression	miniLZO	16.30	233,186	323,633
	Finnish	10.40	908,380	2,208,197
Encryption	MISTY1	18.80	1,267,162	2,086,681
	RC6	11.40	863,348	1,272,845
Data integrity	checksum	9.40	62,560	86,211
	CRC32	9.30	418,598	918,872
Real applications	motion	9.44	3,038,032	4,753,084
	DMU4	19.50	36,808,080	43,186,673
	DMU3	19.59	75,344,906	107,301,464
Stressmarks	stressmotion	9.40	288,745	455,855
	stressDMU3	19.52	124,212	224,791

(*) Typical $10-KB$ workloads have been used, except for DMU-variants which use their own, special workloads.

performing similar functionality in ImpBench, benchmarking diversity has been sought for capturing different processor design aspects. This diversity has already been illustrated in [9] and will be further elaborated in section V. In this version of ImpBench (v1.1), *real applications* have been expanded with *"DMU3"* and a new category *stressmarks* has been added, featuring *"stressmotion"* and *"stressDMU3"*. The original, modified and extended components of the ImpBench benchmark suite are reproduced below.

MiniLZO (shorthand: *"mlzo"*) is a light-weight subset of the LZO library (LZ77-variant). LZO is a data compression library suitable for data de-/compression in real-time, i.e. it favors speed over compression ratio. LZO is written in ANSI C and is designed to be portable across platforms. MiniLZO implements the LZO1X-1 compressor and both the standard and safe LZO1X decompressor.

Finnish (shorthand: *"fin"*) is a C version of the Finnish submission to the Dr. Dobb's compression contest. It is considered to be one of the fastest DOS compressors and is, in fact, a LZ77-variant, its functionality based on a 2-character memory window.

MISTY1 (shorthand: *"misty"*) is one of the CRYPTREC-recommended 64-bit ciphers and is the predecessor of KASUMI, the 3GPP-endorsed encryption algorithm. MISTY1 is designed for high-speed implementations on hardware as well as software platforms by using only logical operations and table lookups. MISTY1 is a royalty-free, open standard documented in RFC2994 [20] and is considered secure with full 8 rounds.

RC6 (shorthand: *"rc6"*) is a parameterized cipher and has a small code size. RC6 is one of the five finalists that competed in the AES challenge and has reasonable performance. Further, Slijepcevic et al. [21] selected RC6 as the algorithm of choice for WSNs. RC6-32/20/16 with 20 rounds is considered secure.

Checksum (shorthand: *"checksum"*) is an error-detecting code that is mainly used in network protocols (e.g. IP and TCP header checksum). The checksum is calculated by adding the bytes of the data, adding the carry bits to the least significant bytes and then getting the two's complement of the results. The main advantage of the checksum code is that it can be easily implemented using an adder. The main disadvantage is that it cannot detect some types of errors (e.g. reordering the data bytes). In the proposed benchmark, a 16-bit checksum code has been selected which is the most common type used for telecommunications protocols.

CRC32 (shorthand: *"crc32"*) is the Cyclic-Redundancy Check (CRC) is an error-detecting code that is based on polynomial division. The main advantage of the CRC code is its simple implementation in hardware, since the polynomial division can be implemented using a shift register and XOR gates. In the proposed benchmark, the 32-degree polynomial[3] specified in the Ethernet and ATM Adaptation Layer 5 (AAL-5) protocol standards has been selected (same as in NetBench)[4].

Motion (shorthand: *"motion"*) is a kernel based on the algorithm described in the work of Wouters et al. [22]. It is a motion-detection algorithm for the movement of animals. In this algorithm, the degree of activity is actually monitored rather than the exact value of the amplitude of the activity signal. That is, the percentage of samples above a set threshold value in a given monitoring window. In effect, this motion-detection algorithm is a smart, efficient, data-reduction algorithm.

DMU4 (shorthand: *"dmu4"*), formerly known as *DMU*[5], is a real program based on the system described in the work of Cross et al. [6]. It simulates a drug-delivery & monitoring unit (DMU). This program does not and cannot simulate all real-time time aspects of the actual (interrupt-driven) system, such as sensor/actuator-specific control, low-level functionality, transceiver operation and so on. Nonetheless, the emphasis here is on the operations performed by the implant core in response to external and internal events (i.e. interrupts). A realistic model has been built imitating the real system as closely as possible.

DMU3 (shorthand: *"dmu3"*) is an extension of *"dmu4"*. The original *"dmu4"* benchmark emulates a real-time implantable system by reading real pressure, temperature and integrated-current sensory data (as provided by true field measurements by the implant developers [6]) and by writing to transceiver module (abstracted as a file). *"dmu3"* emulates this in a more sophisticated manner by also accurately modeling the gascell unit used to switch drug delivery in the implant on and off. To this end, it reads additional input data termed "gascell override switch" and "gascell override value". The suffix numbers in both *DMU* benchmarks originate from different field-test runs using different drug-delivery profiles: A high-low-high varying, "bathtub" profile (#4) has been used for *"dmu4"* and a constant, flat profile (#3) has been used for *"dmu3"*. Due to its affinity with *"dmu4"*, *"dmu3"* will not be analyzed further in this paper. It has been briefly introduced,

[3]CRC32 generator polynomial: $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$.

[4]Erratum correction: In both the original and current ImpBench paper, the standard Ethernet CRC32 has been used, although erroneously reported as being ITU-C CRC16. Reasons for maintaining this CRC32 version here include the fact that CRC16 is too weak to guarantee data integrity in mission-critical applications as implants are.

[5]The original benchmark DMU has been renamed to DMU4, to differentiate it from the new addition DMU3. See main text for further details.

TABLE II
ARCHITECTURAL DETAILS OF (MODIFIED) XTREM.

Feature	**Value**	**Feature**	**Value**
ISA	32-bit ARMv5TE-compatible	Ret. Address Stack	VAR size
Pipeline depth / width	7/8-stage, super-pipelined / 32-bit	I/D TLB (separ.)	VAR size / VAR size
RF size	16 registers	Write Buf. / Fill Buf.	VAR size / VAR size
Issue policy / Instr. Window	in-order, single-instruction	Mem. bus width	1B (1 mem. port)
I/D Cache L1 (separ.)	VAR** size/assoc. (1-cc hit / 170-cc miss lat.)	INT/FP ALUs	1/1
BTB	VAR** size, fully-assoc. / direct-mapped	Clock frequency	2 MHz
Branch Predictor	VAR** (4-cc mispred. lat.)	Implem. Technology	0.18 μm @ 1.5 Volt

(**) Values denoted with 'VAR' indicate adjustable parameters by the GA. For complete parameter ranges refer to [13].

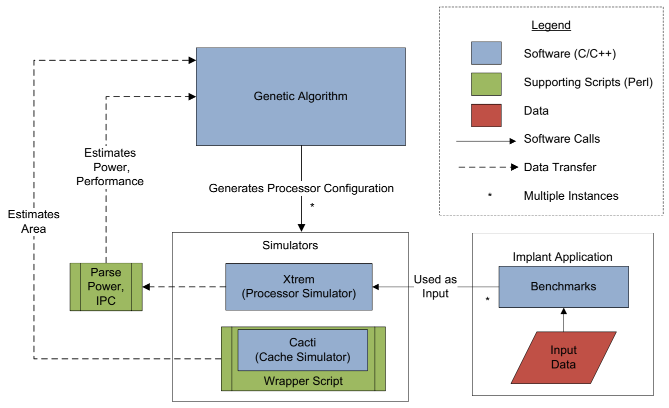


Fig. 2. Framework organization.

though, so that the content of stressmark *"stressdmu3"* will be better understood.

Stressmotion (shorthand: *"stressmotion"*) and **stressDMU3** (shorthand: *"stressdmu3"*) constitute a new addition to the ImpBench suite, their creation stemming from the fact that the original *"motion"* and *"dmu4"* benchmarks have considerably long run times w.r.t. the rest of the benchmarks (see Table I). *"dmu3"* rather than *"dmu4"* has been used for extracting a stressmark due to its more sophisticated emulation of the DMU applications. Since all benchmarks essentially are pieces of continuously iterated code, each stressmark in fact is a derived, worst-case iteration of its respective benchmark. That is, an iteration wherein the implant is required to perform all possible operations; thus, the term "stressmark". As we will see in the following analysis, the stressmarks feature significantly shorter execution times.

With the exception of the DMU-based benchmarks that use their own internal input data, all other benchmarks come with a full complement of physiological *workloads* (e.g. EEG, EMG, blood pressure, pulmonary air volume). Without loss of generality, for this paper we have selected and executed only the $10-KB$ EMG workload ("EMGII_10.bin", see [23] for details) as it exhibits worst-case performance characteristics and, thus, provides a lower-bound for processor design.

IV. Experimental Setup

As evaluation framework for our characterization, we have employed ImpEDE, a previously proposed, GA-based, multiobjective, DSE framework for investigating Pareto-optimal implant-processor alternatives [13]. An overview of the framework is shown in Fig. 2. Optimization (minimization) objectives within the framework are: *processor performance* (in CPI), *processor total area utilization* (in mm^2) and *processor average power consumption* (in mW). As shown in the same figure, a well-known GA (NSGA-II [24]) has been selected for traversing the design space. It generates valid processor configurations also known as "chromosomes". Compromising between unrealistic execution times and quality of results, all full runs of the GA have been allowed to evolve for 200 generations with a population size of 20 chromosomes per generation.

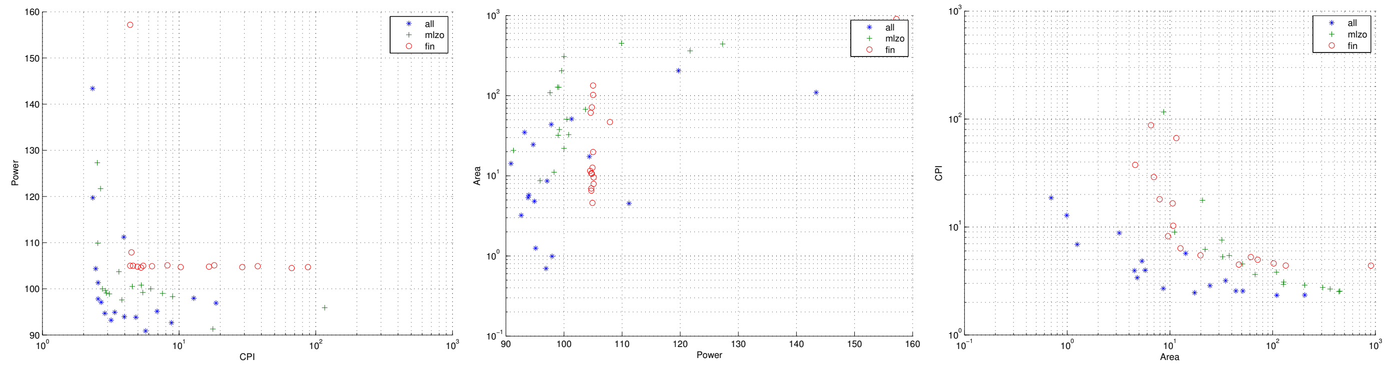
Within the framework, chromosome performance and power metrics are provided by executing the ImpBench benchmarks on the XTREM processor simulator [25]. XTREM is a cycle-accurate performance and power simulator of Intel's XScale embedded processor. It allows monitoring of 14 different subsystems, the most pertinent to our study being: Branch-Target Buffer (BTB), Instruction Cache (I$), Data Cache (D$), Internal Memory Bus (MEM) and Memory Manager (MM). While we have kept some XTREM parameters fixed in order to model implant processors more accurately, we have purposefully left some others variable for the GA to explore their optimal settings, as summarized in Table II. For quantifying each chromosome's area cost, we have used CACTI v3.2, a well-known, cache-area estimation tool.

ImpEDE has primarily been used for exploring promising implant-processor configurations. However, in this work we employ it as a means of *characterizing* the various ImpBench benchmarks in terms of the different directions they push the GA-based optimization process. In short, we wish to compare the Pareto-optimal fronts of the ImpBench benchmarks and to identify differences in resulting processor configurations.

V. Benchmark Characterization

The first characterization study of ImpBench [9] has focused on illustrating its novelty and variation - thus, its significance - with respect to the most closely related MiBench suite. In the following analysis, we investigate further important attributes within the updated suite. Namely, we address the below questions:

(a) What is the aggregate Pareto front of optimal processor configurations, as driven by the whole ImpBench suite? What are the implications in predicted processor-hardware resources?



(a) CPI (-) – Power consumption, avg. (mW) (b) Power consumption, avg. (mW) – Area (mm^2) (c) Area (mm^2) – CPI (-)

Fig. 3. Final Pareto-fronts (after 200 generations) for lossless-compression benchmarks on $10-KB$ workloads.

(b) What is the contribution to the aggregate Pareto front of each separate benchmark? What is the contribution to the predicted hardware resources?

(c) What is the complexity (in simulation time) of ImpBench as a whole and of its components? With respect to the previous questions, can a representative ImpBench subset with significantly shorter simulation times be identified?

A. Lossless compression

Each complete run of our DSE framework evolving chromosomes for 200 generations, results in - at most - 20 Pareto-optimal, implant-processor configurations. Three-dimensional plots can be produced, revealing the shape of the true[6] Pareto curve. Figure 3 illustrates, across the three objectives, three such fronts: the aggregate Pareto front P^{*}[7] labeled in the plots as *"all"* and used as the reference front, and two Pareto fronts formed when only *"mlzo"* and *"fin"* benchmarks are used for the GA evolutions.

Figure 3a depicts clear power and performance cut-offs for all three cases with a wide dispersion of chromosomes, indicating a very well-defined design space. We can observe that both compression algorithms achieve a quite distributed performance-power front, yet *"mlzo"* is closer to the aggregate *"all"* than *"fin"* and is, thus, a better candidate for substituting *"all"*. Figure 3b reveals that, in terms of area, *"mlzo"* is also closer matching *"all"*, even though it appears to be having slightly higher area requirements.

To better understand what these area requirements might translate to in a real processor core, we have put together Fig. 4. In this Figure, boxplots are drawn for different subsystems of the explored processors. Each boxplot has been created by either running the GA with a single benchmark or the whole ImpBench. Statistics (min, max, median etc.) have been calculated based on the evolved population of 20 processor solutions that reside on the Pareto front. In the current analysis, we will focus on a limited number of observations from this Figure. However, Fig. 4 contains a large amount of information and can offer the interested reader more predictions on the various processor during architectural exploration.

For the case of lossless-compression benchmarks, Fig. 4 reveals that *"mlzo"* tends to lead to processors with slightly higher provisions, in particular in the L1 D-cache (D$, hereon) subsystem compared to *"all"* and *"fin"*. For instance, *"mlzo"* requires, on average, a D$ of $64\ KB$ size[8] compared to $4\ KB$ for *"fin"* and $8\ KB$ for *"all"* (see Figs. 4h, 4i and 4j). By observing the rest of the plots in Fig. 4, it becomes apparent that, in an overall, *"mlzo"* is very close (slightly worse) to *"all"* in terms of performance and power but - if it substituted the whole ImpBench in the processor DSE - it would lead to more area-hungry processor configurations. Therefore, *"mlzo"* would be an interesting replacement for ImpBench, always giving worst-case design boundaries. *"fin"*, on the other hand, is more obscure in this respect requiring, on average, smaller D$ but larger BTB structures than *"all"*.

All Pareto fronts Q evolved from single-benchmark runs, present similar (but not identical) 3D-plots. Since results are numerous, we have chosen to also use *arithmetic metrics* to evaluate the benchmarks in a more quantitative and, thus, more reliable manner. For quantifying the **distance** between each single-benchmark Pareto front Q and the reference (aggregate) Pareto front P^{*}, we have chosen to use Veldhuizen's Generational-Distance (GD) metric [26]:

$$\text{Let} \qquad d_i = \min_{k=1}^{|P^*|} \sqrt[p]{\sum_{m=1}^{\mathbb{M}} ({f_m}^{(i)} - {f^*}_m^{(k)})^p}$$

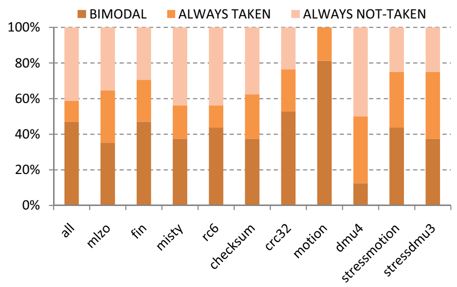
$$\text{Then,} \qquad GD = \frac{\left(\sum_{i=1}^{|Q|} {d_i}^p\right)^{1/p}}{|Q|},$$

where Q is the solution under consideration; P^{*} is the reference Pareto-front, $\mathbb{M}$ is the total number of objective functions,

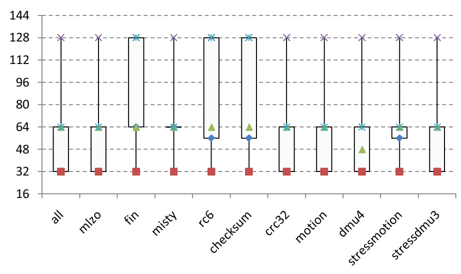
[6]In a real-world optimization problem like ours, the true Pareto front is not known. Therefore, we make the reasonable assumption (and have verified to the best of our equipment's capabilities in [13]) that the aggregate front P^{*} reached after 200 generations matches the true Pareto front P,i.e. $|P^{*} - P| \approx 0$. This means that the Pareto front at generation 200 is considered our reference front for comparisons.

[7]The aggregate Pareto front P^{*} has resulted from running the GA with all original ImpBench benchmarks. That is, *"dmu3"* and both stressmarks are excluded, since they are covered by *"dmu4"*.

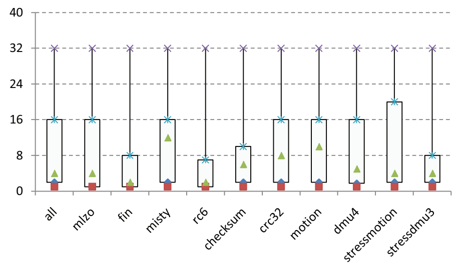
[8]It holds that: $cachesize = \#sets * blocksize * associativity$.



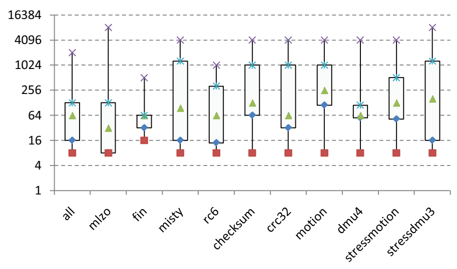
(a) BPRED policy (NT/T/bimodal)



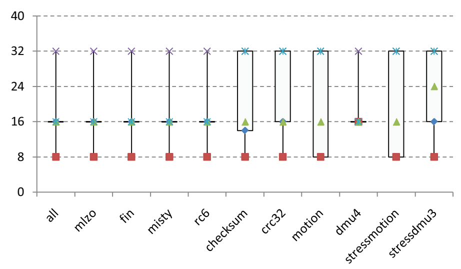
(b) BTB sets (#)



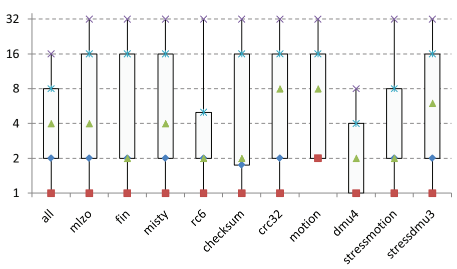
(c) BTB associativity (#ways)



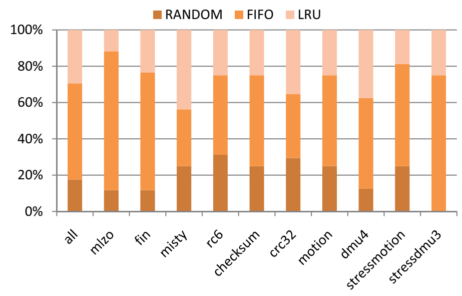
(d) L1 I-cache sets (#)



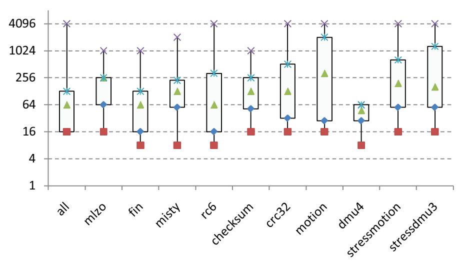
(e) L1 I-cache block size (Bytes)



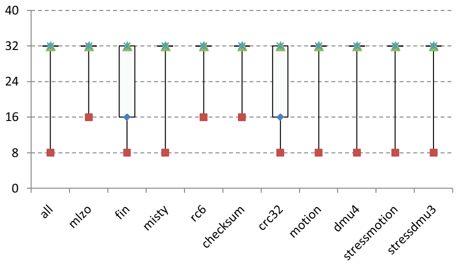
(f) L1 I-cache associativity (#ways)



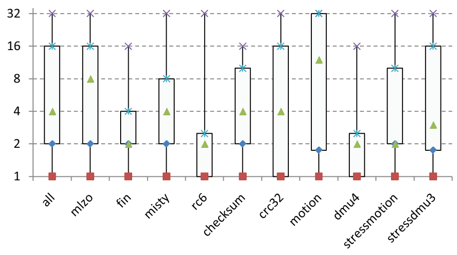
(g) L1 I-cache replacement policy (LRU/FIFO/random)



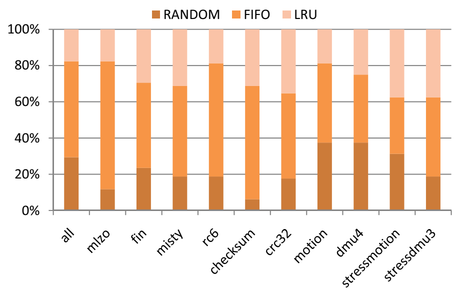
(h) L1 D-cache sets (#)



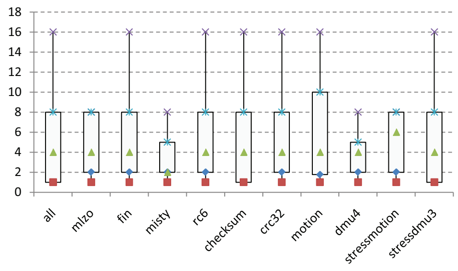
(i) L1 D-cache block size (Bytes)



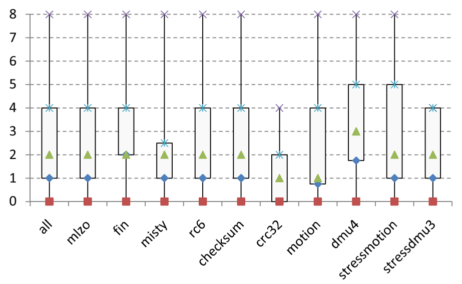
(j) L1 D-cache associativity (#ways)



(k) L1 D-cache replacement policy (LRU/FIFO/random)

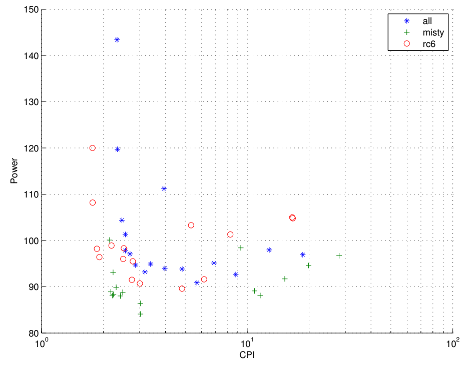


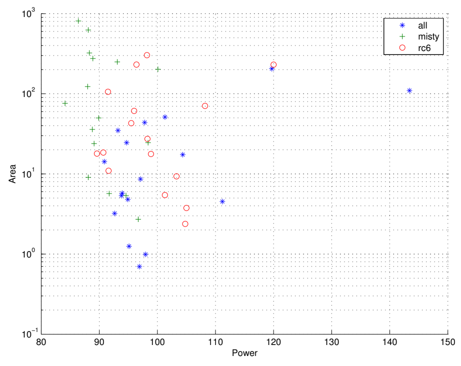
(l) L1 D-cache miss latency (#c.cycles)



(m) Return-address stack (RAS) (#entries)

Fig. 4. Hardware requirements of optimal processors, as evolved over 200-generation runs. Each boxplot is a separate GA run either for a single benchmark or for the whole ImpBench suite and contains statistical results of 20 processors, at most. Barchars are plotted for (a), (g) and (k) since data in these cases are non-numerical. Legend: Medians (triangle), 1st and 3rd quartiles (star and rhombus), min (square), max (x-symbol).


(a) CPI (-) – Power consumption, avg. (mW)


(b) Power consumption, avg. (mW) – Area (mm^2)

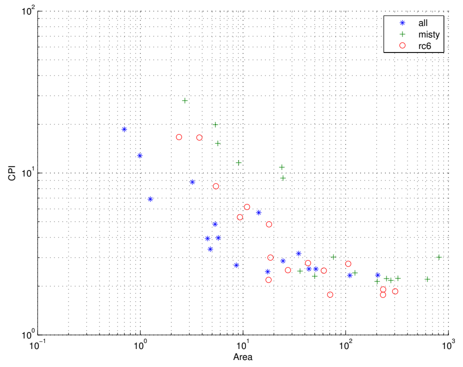

(c) Area (mm^2) – CPI (-)

Fig. 5. Final (after 200 generations) Pareto-fronts for symmetric-encryption benchmarks on $10 - KB$ workloads.

TABLE III
PARETO-FRONT DISTANCE AND NORMALIZED-SPREAD METRICS AND AVERAGE SIMULATION TIME PER BENCHMARK.

Benchmark	GD	Δ (normalized)	Sim. time*** (average) (sec)
miniLZO	0.082	0.383	3.07
Finnish	0.115	0.367	21.82
MISTY1	0.083	0.181	16.65
RC6	0.049	0.151	9.37
checksum	0.090	0.189	0.80
CRC32	0.111	0.285	5.46
motion	0.094	0.163	31.16
DMU4	0.085	0.174	37.25
stressmotion	0.088	0.266	1.33
stressDMU3	0.105	0.143	0.60
all	**0.000**	**0.000**	**125.58**

(***) Measured on a dual-core, AMD Athlon(TM) XP 2400+ @ 2000.244 MHz, cache size 256 KB running Fedora 8 Linux.

and ${f_m}^{(x)}$ and ${f^*{}_m}^{(x)}$ are the m^th^-objective-function values of the x^th^ solutions of Q and P^*, respectively. The lower the value of GD, the closer the distance between Q and P^* and the better the solution, with $GD = 0$ for solutions lying on the reference front.

For quantifying **diversity**, we have used Deb et. al's spread metric Δ [27]:

$$\Delta = \frac{\sum_{m=1}^{\mathbb{M}} d_m^e + \sum_{i=1}^{|Q|} |d_i - \bar{d}|}{\sum_{m=1}^{\mathbb{M}} d_m^e + |Q|\bar{d}}$$

where d_i is the same distance metric described in GD, and $\bar{d}$ is their mean and d_m^e is the distance between the extreme solutions of P^* and Q with respect to the mth objective.

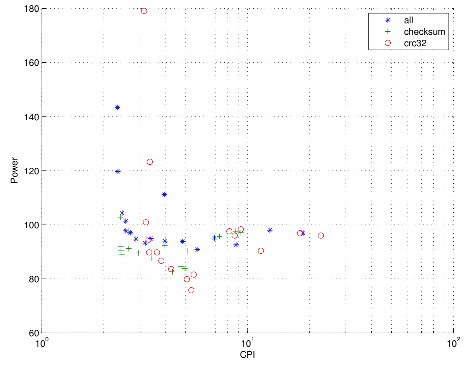
Distance and spread calculations for each benchmark are accumulated in Table III. Numbers verify the visual observations we have made based on Fig. 3. The sharper matching of *"mlzo"* to *"all"*, as compared with *"fin"*, is revealed here by the distances of the two compression benchmarks; 0.082 and 0.115, respectively. In fact, *"mlzo"* features the second smallest GD to *"all"* after *"rc6"*, to be discussed next.

The spread, Δ, of the compression benchmarks, though, is the worst across ImpBench and is slightly better for *"fin"* than it is for *"mlzo"* (0.367 and 0.383, respectively), which is especially discernible in Figs. 3b and 3c. Wider spreads (i.e. smaller Δ's) should imply a wider choice of (optimal) processor configurations from which to pick, as shown in Fig. 4. From the same Figure we can see that, for the two compression algorithms, BTB sets and BTB associativity vary inversely, resulting in the same overall range of BTB sizes. However, as boxplots in the Figure reveal, *"fin"* displays significantly wider ranges in D$ sizes than *"mlzo"*. This difference could account for the difference in Δ's since all other range differences between the two benchmarks are small.

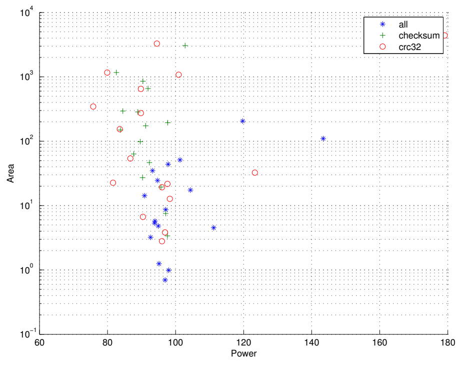
B. Symmetric encryption

In Fig. 5 are plotted aggregate, *"misty"* and *"rc6"* Pareto fronts. Although results are more "noisy" than in the case of the compression benchmarks, it is clear that both encryption benchmarks are closer to *"all"*, with *"rc6"* practically being on top of it. This is supported by the GD values in Table III which reveals that, in fact *"rc6"* and *"misty"* respectively display the 1st and 3rd smallest GD's overall. Inspection of all three plots of Fig. 5 further reveals that *"misty"* consumes less power but requires more area than *"rc6"* (and, thus, *"all"*) while scoring better performance than either of them. This agrees with existing literature [23] and with the targeted applications of MISTY1 which are low-power, embedded systems. Its increased area requirements w.r.t. *"all"* are manifest in Table 4 across the set and associativity (median) sizes of both the I$ and D$ caches. On the other hand, *"rc6"* features slightly reduced area w.r.t. *"all"*, mainly due to a lower (median) associativity degree in both cache structures.

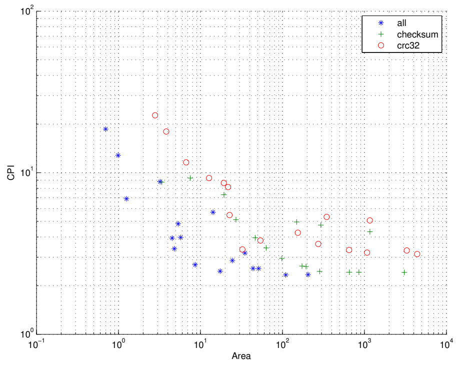
In terms of diversity of solutions, *"misty"* is somewhat more clustered than *"rc6"* (see Fig. 5a) and expectedly has a somewhat larger Δ. Yet, both benchmarks display good spreads with *"rc6"* ranking overall 2nd best. This essentially means that a number of diverse (optimal) processor configurations exists for servicing either benchmark. In particular *"rc6"* is a highly scalable application, which can lead to diverse processors while maintaining low area requirements (as seen above). To be precise, Fig. 4 reveals that *"rc6"* leads to



(a) CPI (-) – Power consumption, avg. (mW)

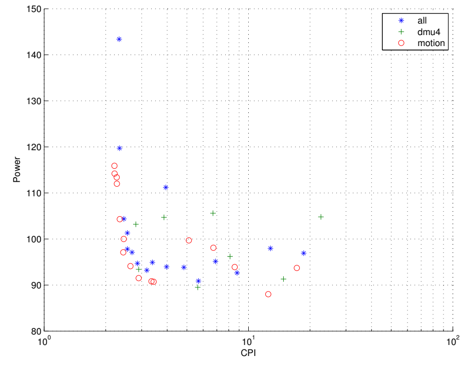


(b) Power consumption, avg. (mW) – Area (mm^2)

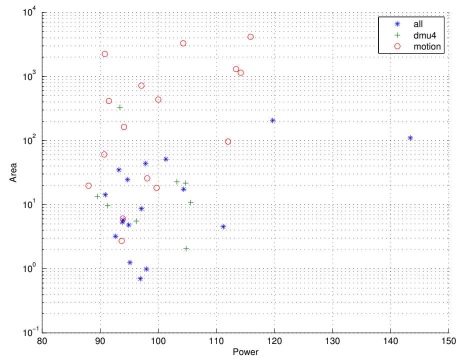


(c) Area (mm^2) – CPI (-)

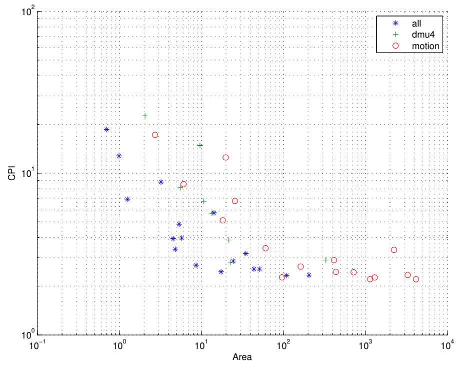
Fig. 6. Final (after 200 generations) Pareto-fronts for data-integrity benchmarks on $10 - KB$ workloads.



(a) CPI (-) – Power consumption, avg. (mW)



(b) Power consumption, avg. (mW) – Area (mm^2)



(c) Area (mm^2) – CPI (-)

Fig. 7. Final (after 200 generations) Pareto-fronts for real applications on $10 - KB$ workloads.

processors with at least $\times 4$ smaller BTB, I\$ and D\$ structures without sacrificing processor flexibility. For instance, the D\$-set boxplot (Fig. 4h) of *"rc6"* indicates most popular sizes from 16 to 64 entries.

In terms of D\$ miss latency, *"rc6"* is also more relaxed compared to *"misty"*. It allows for latencies up to 8 *cycles* (which is similar to what the aggregate run indicates) while *"misty"* can tolerate maximum miss penalties of 5 *cycles* (median). That *"misty"* can lead to overestimations in latency (and overall performance) can also be clearly seen by its increased CPI w.r.t. *"all"* in Fig. 5c. To sum up, for the case of the encryption benchmarks, if *"misty"* was selected alone to drive the exploration process, it would underestimate performance and power costs and overestimate area costs.

C. Data integrity

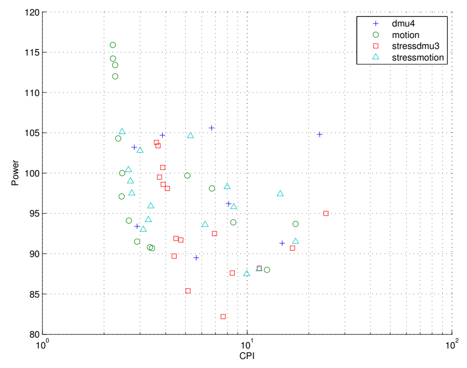
"checksum" and *"crc32"* Pareto fronts are illustrated, along with aggregate, *"all"* Pareto fronts in Fig. 6. Compared to *"all"*, *"checksum"* is somewhat slower and requires more area. *"crc32"* is even slower and results in processor-area costs $\times 2$ those of *"all"*. As an indication of the different area ranges involved, from Figs. 4b and 4c, *"all"* leads the exploration to a median BTB size of 2 KB while *"checksum"* to 3 KB and *"crc32"* to 4 KB. These observations are corroborated by the GD of *"checksum"* being equal to 0.090 and that of *"crc32"* being worse and equal to 0.111.

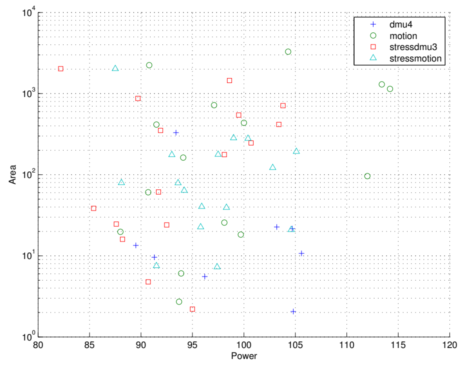
"crc32" is also more irregularly distributed than *"checksum"*, therefore its Δ (0.285) is much worse than that of *"checksum"* (0.189) which is following *"all"* more closely. Similarly to *"rc6"*, *"checksum"* offers a wide spectrum of processor alternatives, while *"crc32"* results in more clustered solutions. It should be noted that this clustering is not always a downside of a benchmark, especially in cases where we are interested in neighboring alternatives in the same design niche. It offers finer resolution within the area of interest.

The proximity of *"checksum"* to *"all"* is also reflected in Fig. 4 on the identical boxplots for the D\$ miss latency and the RAS size. Yet, *"checksum"*'s simpler structure seems to favor the simpler branch-prediction technique ALWAYS TAKEN while the more complex *"crc32"* tends towards the more powerful BIMODAL predictor. Overall, *"checksum"* appears capable of substituting *"all"* in simulations but we should always account for the observed increase in area and decrease in performance of the resulting processor configurations. *"crc32"*, on the other hand, features the second worst GD after *"fin"* and its Δ is mediocre. Combined with its high area costs, it should not be considered in most cases as a good, single substitute for *"all"*.

D. Real applications & stressmarks

In Fig. 7, Pareto fronts for the real applications *"motion"* and *"dmu4"* are compared with *"all"*. The plots reveal good fitting and dispersal of the solutions for both benchmarks, and GD/Δ figures agree with these observations. Power and


(a) CPI (-) – Power consumption, avg. (mW)


(b) Power consumption, avg. (mW) – Area (mm^2)

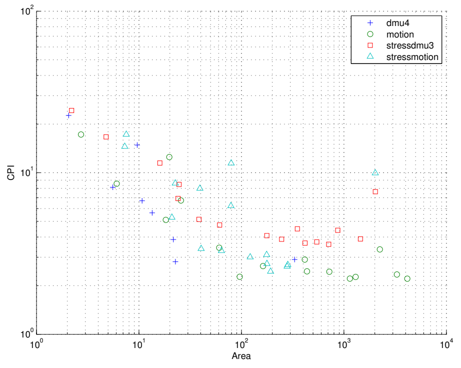

(c) Area (mm^2) – CPI (-)

Fig. 8. Final (after 200 generations) Pareto-fronts for real applications and stressmarks on their respective workloads.

performance ranges are similar to *"all"*, yet area ranges are significantly different. Although *"motion"* has a much simpler functionality than *"dmu4"*, it incurs disproportional area costs: on average 32 KB for the BPRED/BTB, 120 KB for the D\$ and 5 KB for the I\$, as opposed to *"dmu4"* which promotes processors with small average sizes: 2 KB, 3 KB and 1.8 KB, respectively.

Regarding branch-prediction hardware, *"motion"* presents unexpected results, too. It almost exclusively favors configurations equipped with a BIMODAL branch predictor. Conversely, *"dmu4"* opts mainly for the simpler static predictors ALWAYS TAKEN and ALWAYS NOT-TAKEN. On the other hand, *"dmu4"* appears to suffer more from increased D\$ miss latencies, compared to the average case. In effect, *"all"* evolves processor configurations with latencies up to 8 cycles while *"motion"* drives latencies up to 10 cycles and *"dmu4"* only up to 5 cycles. As opposed to the preceding benchmarks, in this case, *"motion"* and *"dmu4"* display diverse properties which cannot be covered fully by either single one of them. This is to be expected for benchmarks (in essence, kernels) emulating implant applications. If one real benchmark had to be selected as representative for this group, *"dmu4"* would be the safer choice.

In order to compare the characteristics of the real benchmarks, above, and the newly-created stressmarks, we have plotted Fig. 8, where the Pareto fronts of all four programs are being shown. Plot 8a reveals that, in terms of performance and power, *"stressmotion"* has a close distance to *"motion"* albeit a slightly worse spread. In terms of hardware requirements, *"stressmotion"* relaxes design requirements compared to *"motion"*: BPRED/BTB 4 KB, D\$ 12 KB and I\$ 2 KB on average. In an overall, though, by combining *"stressmotion"* from Fig. 8 with *"all"* from Fig. 7, we can see that *"stressmotion"* is actually closer to the aggregate Pareto line than *"motion"*, as verified by the respective GD's. Essentially, *"stressmotion"* can track the Pareto front better than the full *"motion"* benchmark, albeit with somewhat more clustered solutions.

"stressdmu3", on the other hand, follows *"all"* with somewhat less fidelity than *"dmu4"* but displays a better spread. As Fig. 8 indicates, both *"stressdmu3"* and *"dmu4"* fronts are residing in the same locus of solutions. Yet, *"stressdmu3"* achieves slightly lower performance and drives processor resources slightly up compared to *"dmu4"*, as follows: BPRED/BTB 22.5 KB, D\$ 15 KB and I\$ 2 KB on average.

As far as branch-prediction requirements are concerned, GA evolutions reveal the following: As stressmarks run for a short period of time (one or a few iterations only), branch-predictor distributions in the barcharts of Fig. 4a are relatively unaffected. The simpler scheme ALWAYS TAKEN is mostly expanded but, other than that, both stressmarks bear distributions similar to *"all"*.

Conclusively, the new stressmarks track the true Pareto front closely, each either scoring better in GD or Δ. It is interesting to notice that, while they have not contributed to the true Pareto front (i.e. they have not been used in the aggregated-benchmark simulations), the solutions they evolve are highly comparable and in the same locus as the aggregate solutions. This fact attests to their prediction quality and, provided that attention is paid to the differences they exhibit (as discussed above), they can offer reliable substitutes for the full real applications.

VI. Discussion

The previous analysis has unveiled new information with respect to the coverage of the design space and the hardware implications contributed by each ImpBench component. In this context, results indicate that, from the lossless-compression benchmarks, *"mlzo"* provides better GD and similar Δ to *"fin"*. On top of this, it also features faster simulation times (about $\times3$ faster, according to Table III). From the symmetric-encryption benchmarks, *"rc6"* displays excellent characteristics, namely the smallest GD and the second best Δ, meaning that it traces the aggregate Pareto front with high fidelity. According to Table III, it is also about double as fast as *"misty"*. Of the data-integrity benchmarks, *"checksum"* performs consistently better than *"crc32"* and features the overall shortest simulation time (0.80 sec) across all full benchmarks.

For the real-applications case, no single benchmark could be unanimously ranked higher due, mainly, to the complex nature of the results. This actually shows the usefulness of both real benchmarks which, also, feature similar simulation times (and the largest in the whole suite). Last, analysis of the new stressmarks has revealed that, although they display some variability in predicted processor specifications w.r.t. the full real applications, they both track the true Pareto front closely. Careful use of the stressmarks can seriously reduce simulation times up to $\times 30$ (see Table III), which is an impressive speedup and a good tradeoff between DSE speed and accuracy.

The above results indicate highest-ranking benchmarks within the ImpBench suite, however this is not to say that the poorest-performing ones are redundant. The findings of the original analysis [9] indicate that each benchmark in ImpBench exhibits diverse characteristics (e.g. μop mixes) and should not be dropped from consideration when considering implant-processor design. On the contrary, this study comes as a complement and extension of the original ImpBench study.

VII. Conclusions

In view of more structured and educated implant processors in the years to come, we have carefully put together ImpBench, a collection of benchmark programs and assorted input datasets, able to capture the intrinsic of new implant architectures under evaluation. In this paper, we have extended the suite with an additional benchmark and two stressmarks. The stressmarks proposed, feature fraction-of-the-original-time execution times and, provided that their peculiarities are taken into account by the designer, they can considerably enhance and speedup processor-profiling times. We have, further, expanded our characterization analysis to include predictions on actual implant-processor configurations resulting from the use of this suite as a profiling basis. ImpBench is a dynamic construct and, in the future, more benchmarks will be added, subject to our ongoing research. Among others, we anticipate simple DSP applications as potential candidates as well as more real applications like the "artificial pancreas", a crucial application nowadays for diabetic patients.

VIII. Acknowledgements

This work has been partially supported by the ICT Delft Research Centre (DRC-ICT) of the Delft University of Technology. Many thanks are due to Erik de Vries and Eef Hartman for their excellent technical support in setting up and operating the DSE computer cluster. Without them, this work would not have been possible.

References

[1] A. Omran, "The epidemiological transition," *Milbank Memorial Fund Quarterly*, vol. 49, no. 4, pp. 509–538, 1971.

[2] R. Beaglehole and R. Bonita, *Public health at the crossroads: achievements and prospects*, 2nd ed. Cambridge University Press, 2004.

[3] H. Ector and P. Vardas, "Current use of pacemakers, implantable cardioverter defibrillators, and resynchronization devices: data from the registry of the european heart rhythm association," *European Heart Journal Supplements*, vol. 9, pp. 144–149, August 1988.

[4] F. Nebeker, "Golden accomplishments in biomedical engineering," in *IEEE Engineering in Medicine and Biology Magazine*, vol. 21, Piscataway, NJ, USA, May - June 2002, pp. 17–47.

[5] C. Liang, J. Chen, C. Chung, C. Cheng, and C. Wang, "An implantable bi-directional wireless transmission system for transcutaneous biological signal recording," *Physiological Measurement*, vol. 26, pp. 83–97, February 2005.

[6] P. Cross, R. Kunnemeyer, C. Bunt, D. Carnegie, and M. Rathbone, "Control, communication and monitoring of intravaginal drug delivery in dairy cows," in *International Journal of Pharmaceuticals*, vol. 282, 10 September 2004, pp. 35–44.

[7] H. Lanmuller, E. Unger, M. Reichel, Z. Ashley, W. Mayr, and A. Tschakert, "Implantable stimulator for the conditioning of denervated muscles in rabbit," in *8th Vienna International Workshop on Functional Electrical Stimulation*, Vienna, Austria, 10-13 September 2004.

[8] C. Strydis, G. Gaydadjiev, and S. Vassiliadis, "Implantable microelectronic devices: A comprehensive review," Computer Engineering, Delft University of Technology, CE-TR-2006-01, December 2006.

[9] C. Strydis, C. Kachris, and G. Gaydadjiev, "ImpBench - A novel benchmark suite for biomedical, microelectronic implants," in *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS08)*, 2008, pp. 21–24.

[10] C. Strydis and G. N. Gaydadjiev, "Evaluating Various Branch-Prediction Schemes for Biomedical-Implant Processors," in *Proceedings of the 20th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP'09)*, July 2009, pp. 169–176.

[11] ——, "The Case for a Generic Implant Processor," in *30th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC'08)*, August 2008, pp. 3186–3191.

[12] V. Guzma, S. Bhattacharyya, P. Kellomaki, and J. Takala, "An integrated asip design flow for digital signal processing applications," in *First International Symposium on Applied Sciences on Biomedical and Communication Technologies (ISABEL'08)*, October 2008, pp. 1 –5.

[13] D. Dave, C. Strydis, and G. N. Gaydadjiev, "ImpEDE: A Multidimensional, Design-Space Exploration Framework for Biomedical-Implant Processors," in *Proceedings of the 21th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP'10)*, July 7-9 2010.

[14] "SPEC CPU2006," http://www.spec.org/cpu2006/.

[15] C. Lee, M. Potkonjak, and W. Mangione-Smith, "MediaBench: a tool for evaluating and synthesizing multimedia and communications systems," *30th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 330–335, 1-3 Dec 1997.

[16] "EEMBC," http://www.eembc.com.

[17] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "MiBench: A free, commercially representative embedded benchmark suite," *IEEE International Workshop on Workload Characterization*, pp. 3–14, 2 December 2001.

[18] G. Memik, W. H. Mangione-Smith, and W. Hu, "NetBench: a benchmarking suite for network processors," in *IEEE/ACM international conference on Computer-aided design (ICCAD'01)*, Piscataway, NJ, USA, 2001, pp. 39–42.

[19] T. Wolf and M. Franklin, "CommBench-a telecommunications benchmark for network processors," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'00)*, Washington, DC, USA, 2000, pp. 154–162.

[20] H. Ohta and M. Matsui, *A Description of the MISTY1 Encryption Algorithm*, United States, 2000.

[21] S. Slijepcevic, M. Potkonjak, V. Tsiatsis, S. Zimbeck, and M. Srivastava, "On communication security in wireless ad-hoc sensor networks," *Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE'02)*, pp. 139–144, 2002.

[22] P. Wouters, M. D. Cooman, D. Lapadatu, and R. Puers, "A low power multi-sensor interface for injectable microprocessor-based animal monitoring system," in *Sensors and Actuators A: Physical*, vol. 41-42, 1994, pp. 198–206.

[23] C. Strydis, D. Zhu, and G. Gaydadjiev, "Profiling of symmetric encryption algorithms for a novel biomedical-implant architecture," in *ACM International Conference on Computing Frontiers (CF'08)*, Ischia, Italy, 5-7 May 2008, pp. 231–240.

[24] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A Fast Elitist Multi-Objective Genetic Algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation*, vol. 6, pp. 182–197, 2000.

[25] G. Contreras, M. Martonosi, J. Peng, R. Ju, and G.-Y. Lueh, "XTREM: A Power Simulator for the Intel XScale Core," in *LCTES'04*, 2004, pp. 115–125.

[26] D. A. V. Veldhuizen and G. B. Lamont, "Evolutionary computation and convergence to a pareto front," in *Proceedings of the 3rd Annual Conference on Genetic Programming*, 1998.

[27] K. Deb, *Multi-Objective Optimization Using Evolutionary Algorithms*. John Wiley & Sons, LTD, 2001.

A.3 Identifying optimal, generic, biomedical-implant processors

The following paper has been submitted at ICCD 2010 - 28th IEEE International Conference on Computer Design.

Identifying optimal, generic, biomedical-implant processors

Christos Strydis and Dhara Dave
Computer Engineering Laboratory, Electrical Engineering Dept.
Delft University of Technology
Postbus 5031, 2600 GA, Delft
The Netherlands
C.Strydis@tudelft.nl, D.Dave@student.tudelft.nl

Abstract— **The extremely limited resource budget available to medical implants makes it imperative that they are designed in the most optimal way possible. The limited resources include - but are not limited to - battery life, expected responsiveness of the system and chip area. We have already detailed the design of a design-space exploration (DSE) tool specifically geared towards finding the Pareto-optimal design front. In this paper, we choose processor configurations from the Pareto-optimal processor set found by the DSE using real implants as case studies. We find that even under the extremely biased constraints that we use, our processor(s) perform better than many of the real implants. This provides strong hints towards designing an implant processor with more realistic design considerations that is generic enough to cover most, if not all, implant applications.**

I. Introduction

The market for biomedical-implants is slowly but surely expanding with a rising number of applications. While, at first restricted to the field of pacemakers [2], implants have now diversified and cover nearly all bodily systems, from musculoskeletal, to circulatory, to neural [3]. Moreover, recent trends in global healthcare [4] are pushing towards ”smarter” implants with increased capabilities. An extended study performed on more than 60 different implantable systems attests to this claim [1]. For the 12-year study period 1994 – 2005, Fig. 1a reveals an increasing number of implants charged with non-trivial processing duties and featuring in-system memory blocks. Every year, about 12% more implants perform some complex processing task(s) *in vivo* while 17% more implants are designed with sizeable memories on them.

However, such provisioning comes at a cost. As Fig. 1b reveals, even though operational voltages are dropping in agreement with shrinking process-technology trends, implant power consumption exhibits an aggregate increase of 15%[1].

A third, related trend has also been observed: It has been common practice so far to custom-design the hardware for each implant application, often completely from scratch (see Fig. 1c). Although this was easier for the simpler devices, designing a processing-capable core for every implant application is not practical due to large development and deployment costs. Therefore, the use of commercial, off-the-shelf (COTS) components is also gradually increasing, as witnessed in the same Figure. However, such designs are ad-hoc, are not consistently designed with the restricted resource limitations of implants in mind, as exemplified for power in Fig. 1b, and introduce huge design/testing times and costs.

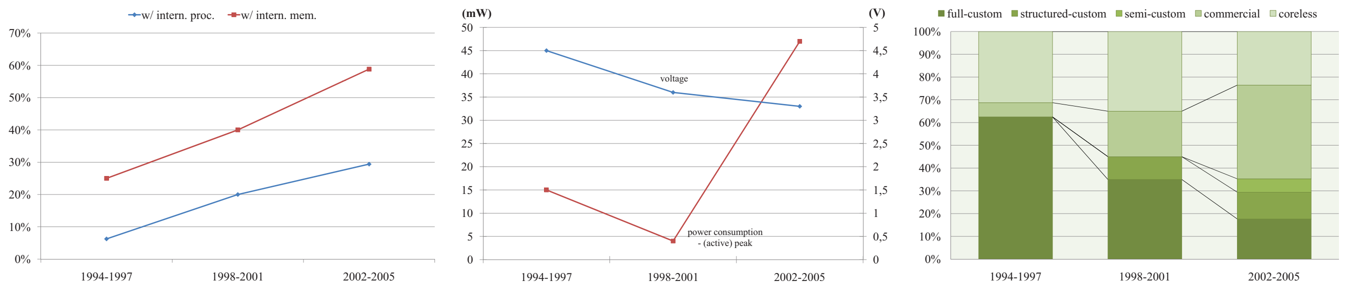
[1]For the observed dip in the middle years 1998 – 2001 a biasing artifact is responsible in the sampled data; see [1] for a detailed explanation.

It is, therefore, becoming apparent that ”smart”, pre-designed and pretested components are needed, that are specifically geared towards medical implants. Such components must cover a *large application range* in order to be economical as well as reliable and safe. This is the express goal of the SiMS project [5]. Our final goal is to design a (so-called SiMS) generic, low-power implant processor or processor family, for covering a large part of the implant domain.

In this paper, we present the results of an automated, design-space-exploration (DSE) effort performed to identify such SiMS processor candidates. We, further, select a number of representative, real implant applications in the literature and explore the possibility of covering them with a few of the identified processors. It should be noted that this study focuses on the microarchitectural aspects of the SiMS processor, thus no Instruction-Set-Architecture (ISA) analysis is present. Concisely, the contributions of this work are:

- To propose Pareto-optimal, alternative microarchitectural configurations for the SiMS processor;
- To make a proof-of-concept, first attempt at fitting a single (or a few) of the identified configurations to real implant applications;
- To propose a new, realistic, worst-case workload mix for future implant processors; and
- Along with the previously generated DSE toolset and the new workload mix, to provide a complete framework enabling the implant designer to make informed decisions about resource allocation for future implant design.

The paper is organized as follows: section II gives an overview of related works in the field. Section III goes through the experimental tools used for enabling this study and the synthesis of the implant workload used. In section IV a concise presentation of the implant study cases is given. Section V contains, in detail, the findings of this work. Overall conclusions and future work are drawn in section VI.



(a) More implants require processing capabilities and memories.

(b) Implants exhibit decreasing voltage but increasing power needs.

(c) Use of commercial components is increasing at the expense of full-custom design.

Fig. 1. Implant trends over the survey period 1994 – 2005 [1].

II. Related work

In the past, a very limited number of attempts has been made to design implants with some degree of modularity for making them capable of adapting to different application scenarios.

Fernald et al. [6], [7] has come up with a modular microprocessor architecture which accepts various peripheral modules such as sensors, actuators and transceivers. Application flexibility is underpinned by a dual ring-bus interconnect linking an arbitrary number of modules to the processing core which is a fully featured 16-bit μP (PERC), based on Hector [8]. Command and data packets, traveling across each bus, have predefined, consistent structures and plugged modules are built to interface to them.

Contrary to the additive nature of the above design, Smith et al. [9], [10] has addressed the problem of flexibility from a subtractive angle. An implantable stimulator device with provisions for a large set of peripherals was designed. Given a specific application, unutilized components of the initial, baseline design are removed, resulting in a reduced system, sewn to the application needs and with lower power/area requirements than those of the base design.

Valdastri et al. [11] has presented a versatile implantable platform that provides multi-channel telemetry of measured biosignals. Its versatility resides in its ability to support different types of sensors and to allow for easy reprogramming so as to fulfill different application requirements. To demonstrate the correctness of the concept, a specific case study is implemented for gastric-pressure monitoring which is a PCB-mounted assembly, supporting up to 3 sensor channels. This implant can transmit digitally modulated data to an external receiver over a wireless link with robust error control.

Last, Salmons et al. [12] has performed a design and comparative study between an ASIC-based and a microcontroller-based microstimulator device for restoring functionality to paralyzed muscles. Analysis has shown that, if carefully designed with low-power modes and checked for software bugs, the latter version is beneficial to the ASIC with respect to development and testing costs.

The work presented here is original in that it attempts to develop a truly generic and low-power processor architecture while at the same time providing the performance needed by current and future applications in the field. A *systematic*, *structured* approach to the problem, supported by the rapid advances of late in microelectronics technology [13], make such a venture finally realistic.

III. Experimental setup

Our work so far has been focused on investigating the design space of implant processors for proposing one or a few processor architectural configurations able to support a number of implant applications. This task is difficult to tackle as we have repeatedly encountered the following problems:

- Implant applications (and their requirements) are very diverse, mirroring the wide range of potential pathoses in the human body. To make matters worse, biomedical implants are a relatively new field, traditionally dominated by a handful of companies which are extremely protective of their product designs. With literature being limited, consensus in the ”application domain” cannot be easily established.
- A systematic approach and established operational parameters for designing processors specifically tailored to implant applications does not yet exist. Thus, a number of educated assumptions is necessitated for introducing boundaries to the design problem.
- Verified tools for modeling the desired processors and exploring the design space are not readily available. The ones used are best-effort ones which introduce accuracy errors and deviations between simulated and actual results. These deviations are not linear and, thus, cannot be easily predicted in advance.

Except, perhaps, for the first item in this list, all above problems are well-known and have already been encountered in other application fields. If we are to attempt a first take on a (few) processor(s) capable of serving a number of implant applications, we need to fill the missing information with some further estimations. However, we will have to ensure that these estimations will be drawn so that the resulting implant-processor architectures are guaranteed to cover the targeted applications under *worst-case conditions*. In effect, we will intentionally overprovision our processor(s) slightly. For this study to become possible, a number of components is required. In the following subsections, we briefly introduce these components along with their capabilities and limitations.

TABLE I

ARCHITECTURAL DETAILS OF (MODIFIED) XTREM.

Feature	**Value**	**Feature**	**Value**
ISA	32-bit ARMv5TE-compatible	Ret. Address Stack	VAR size
Pipeline depth / width	7/8-stage, super-pipelined / 32-bit	I/D TLB (separ.)	VAR size / VAR size
RF size	16 registers	Write Buf. / Fill Buf.	VAR size / VAR size
Issue policy / Instr. Window	in-order, single-instruction	Mem. bus width	1B (1 mem. port)
I/D Cache L1 (separ.)	VAR size/assoc. (1-cc hit / 170-cc miss lat.)	INT/FP ALUs	1/1
BTB	VAR size, fully-assoc. / direct-mapped	Clock frequency	2 MHz
Branch Predictor	VAR (4-cc mispred. lat.)	Implem. Technology	0.18 μm @ 1.5 Volt

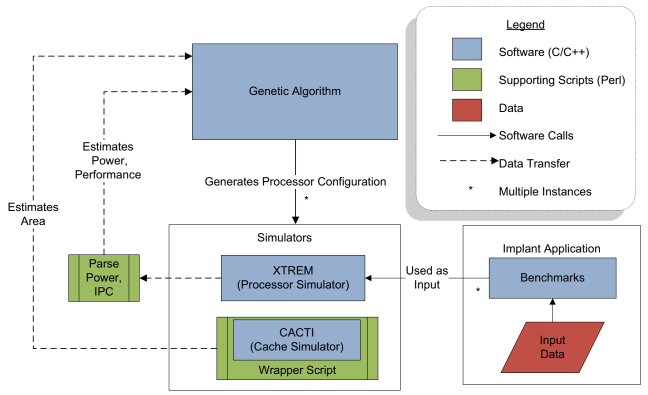


Fig. 2. Overview of ImpEDE exploration framework.

A. Exploration framework

As (automated) exploration tool we have employed ImpEDE, a previously proposed multiobjective, DSE framework for investigating Pareto-optimal, implant-processor alternatives [14]. An overview of the framework is shown in Fig. 2. Optimization (minimization) objectives within the framework are: *maximum execution time* (in sec), *total area utilization* (in mm^2) and *total average power consumption* (in mW). Under 'total', processor as well as (off-chip) memory figures are aggregated. As shown in the same figure, a well-known GA (NSGA-II [15]) has been selected for traversing the design space. It generates valid processor configurations also known as "chromosomes". Compromising between unrealistic execution times and quality of results, all full runs of the GA have been allowed to evolve for 200 generations with a population size of 20 chromosomes per generation.

Within the framework, chromosome performance (i.e. execution time) and power metrics are provided by utilizing XTREM, a cycle-accurate, performance and power XScale-processor simulator [16]. XTREM allows monitoring of 14 different subsystems, the most pertinent to our study being: Branch-Target Buffer (BTB), Instruction Cache (I\$), Data Cache (D\$), Internal Memory Bus (MEM) and Memory Manager (MM). While we have kept some XTREM parameters fixed in order to model implant processors more accurately, we have purposefully left some others variable for the GA to explore their optimal settings, as summarized in Table I[2].

[2]Values denoted with 'VAR' indicate adjustable parameters by the GA. For complete parameter ranges refer to [14]

TABLE II

IMPBENCH V1.1 BENCHMARKS. (*) INDICATES TYPICAL VALUES FOR $10-KB$ WORKLOADS, EXCEPT FOR DMU-VARIANTS WHICH USE THEIR OWN SPECIAL WORKLOADS.

benchmark	**name**	**size (KB)**	**dyn. instr.* (average) (#)**	**dyn. μops* (average) (#)**
Compression	miniLZO	16.30	233186	323633
	Finnish	10.40	908380	2208197
Encryption	MISTY1	18.80	1267162	2086681
	RC6	11.40	863348	1272845
Data integrity	checksum	9.40	62560	86211
	CRC32	9.30	418598	918872
Real applications	motion	9.44	3038032	4753084
	DMU4	19.50	36808080	43186673
	DMU3	19.59	75344906	107301464
Stressmarks	stressmotion	9.40	288745	455855
	stressDMU3	19.52	124212	224791

While XTREM has been very useful in our studies to day, yet it fails in a number of ways, a crucial one being that it models a low-power, high-performance embedded processor, which is kind of "overshoot"; it does not exactly fitting our implant application domain. Another shortcoming is that XTREM does not simulate any (off-chip) memory, thus making system-level simulations difficult. Last, our long usage of XTREM has revealed a number of bugs and modeling inaccuracies[3], most of which have been solved by a newer simulator XEEMU [18]. We are currently busy porting XEEMU to our framework; in the meantime, XTREM has been maintained in our exploration chiefly for reasons of availability and ease of use. We have combined readouts from XEEMU regarding memory power consumption and have updated the power metric in our exploration accordingly.

For quantifying each chromosome's area cost, we have used CACTI v3.2, a well-known, cache-area estimation tool. The total area cost has been calculated by summing the (fixed) net processor and (off-chip) memory area, based on related literature, and the per-case cache (BTB, I\$, D\$ etc.), based on CACTI estimations.

B. Biomedical workload

The last component missing from the above framework is the biomedical workloads that will be fed into the simulators to drive the exploration process. We are essentially interested in benchmarks representative of the actual workloads that will be fed in real implants, in terms of *functional* as well

[3]For an extensive list, see [17].

TABLE III

IMPEDE-EVOLVED PROCESSOR CONFIGURATIONS.

conf.	BPRED	BTB sets	assoc	RAS	L1-I$ sets	bl.size	assoc	repl	L1-D$ sets	bl.size	assoc	repl	Mem lat.	Ex. Time	Power	Area
	(-)	(#)	(#)	(#)	(#)	($bits$)	(#)	(-)	(#)	($bits$)	(#)	(-)	(#cc)	(sec)	(mW)	(mm^2)
1	bimod	64	8	8	4096	16	32	FIFO	4096	16	1	FIFO	2	27.465	17.539	2521.36
2	bimod	128	8	2	256	16	16	LRU	4096	8	2	LRU	16	37.166	15.368	394.53
3	bimod	64	32	0	256	16	16	RAND	1024	32	2	FIFO	1	1.790	123.143	400.92
5	taken			1	1024	32	32	RAND	16	16	16	FIFO	8	26.143	13.842	1325.39
6	nottaken			8	1024	16	4	FIFO	512	32	2	FIFO	1	1.433	63.217	327.10
7	bimod	128	8	4	2048	16	8	FIFO	4096	32	1	LRU	1	1.751	93.200	659.94
8	taken			0	16	32	8	RAND	512	32	4	RAND	8	2.777	74.860	299.37
9	bimod	128	2	4	64	32	8	LRU	128	32	16	FIFO	8	2.181	63.288	327.61
10	bimod	32	16	8	128	8	2	FIFO	16	32	8	RAND	1	4.516	87.887	243.68
11	bimod	64	8	1	256	16	4	FIFO	64	32	16	FIFO	2	1.951	93.366	298.79
12	nottaken			4	16	8	2	FIFO	64	8	2	RAND	8	35.571	88.153	215.30
13	bimod	64	4	2	8	16	1	FIFO	128	32	2	RAND	16	6.834	69.729	227.71
14	nottaken			2	64	16	2	LRU	16	32	16	FIFO	16	4.605	67.197	250.99
15	nottaken			2	16	8	4	FIFO	32	32	2	FIFO	4	6.823	80.947	218.21
16	nottaken			2	8	32	2	LRU	64	16	2	FIFO	1	24.463	71.681	218.84
17	nottaken			1	32	16	16	FIFO	16	32	4	LRU	8	2.868	69.781	238.62
18	bimod	128	2	8	64	8	16	FIFO	16	32	16	FIFO	2	2.222	90.816	268.36
19	bimod	32	1	8	64	16	16	LRU	128	32	32	FIFO	4	1.922	74.419	421.30
20	nottaken			1	128	16	4	LRU	64	32	4	RAND	16	3.395	62.336	236.57

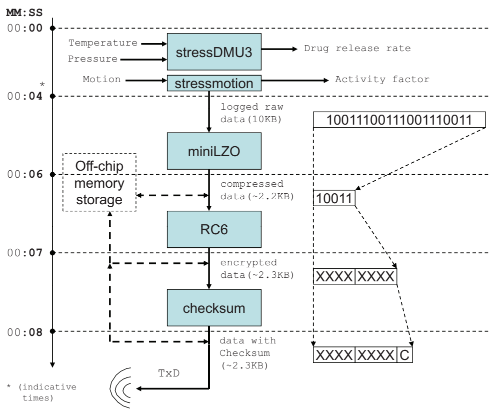


Fig. 3. Conceptual block diagram of simulated implant application (based on [4]).

as of *timing* behavior. To represent these workloads, we make use of the benchmarks found in ImpBench v1.1 [19], comprising compression, encryption, data-integrity, synthetic and stress benchmarks (see Table II).

As indicated also by the benchmarks, implant functionality is a largely iterative process wherein sensors are periodically read, actuators are periodically enabled and processing tasks are periodically triggered. A general, typical workload mix for future implants has already been presented in [4]. Concisely, a synthetic application (DMU-variant) executes (manipulating sensors and actuators) and periodically (when 10 KB of logged data are collected) compression, encryption and data-integrity tasks are invoked on the data.

In this case, and in order to provide realistic, *worst-case*, SiMS-processor design, we update this workload mix as follows: Per benchmark category, we select the fastest executing algorithm - i.e. *miniLZO* for compression, *RC6* for encryption and *checksum* for data integrity. As for the synthetic benchmark, we replace it by *both* stressmarks *stressmotion* and *stressDMU3* which simulate a single-iteration, worst-case instance of the regular benchmarks *motion* and *DMU3*, respectively. This combination of benchmarks is depicted in Fig. 3.

Every processor configuration (or chromosome) evolved through ImpEDE is made to execute this whole sequence of benchmarks, representing *the busiest (i.e. worst-case) iteration* in the implant's operational lifetime. The *execution-time* metric is calculated as the accumulation of execution times of all involved benchmarks while the *power-consumption* metric is calculated as the weighted average of the power consumptions of all involved benchmarks with each one's execution time used as the weighting coefficient.

To push the worst-case, processor-design envelope further, and without loss of generality, we use $10-KB$ EMGII as the input dataset to the above benchmarks. It features a realistic size and has been shown to evoke the longest execution times among the available physiological datasets [14].

It should be noted, last, that all ImpBench benchmarks (and, thus, the ones currently used) are kernels simulating the processing load of an implant processor. Therefore, they suffer from certain modeling limitations: they have no way (a) of modeling the behavior of any implant peripherals (biosensors/bioactuators), and subsequently (b) of accurate modeling any externally triggered (timing or other) events, i.e. they have no sense of real time. This is a well-known problem in benchmarking (event-driven) embedded systems. It can be addressed (and has been, in our case) by introducing extra code in the benchmarks to imitate the passage of time and the occurrence of external events (e.g. timer/sensor interrupt). This, of course, has to be done in a careful fashion as it can potentially pollute simulation results in terms of timing behavior, executed instruction mix and so on.

Under all above considerations, ImpEDE has been allowed to run over significant periods of time in search of optimal SiMS-processor configurations. Results are given in Table III. Each one of the 19 entries is a Pareto-optimal, non-dominated solution to the problem. Performance, power and area metrics are also reported for each entry.

TABLE IV

STUDY CASES OF REAL IMPLANTABLE APPLICATIONS (TAKEN FROM [1]).

case	Author	Pub. Year	Application	Power source (-)	Sensor count (#)	Sampl. rate (Hz)	ADC resol. (bits)	Core arch. (-)	Core freq. (MHz)	**Ex.Time Worst-case (sec)**	**Power Peak (mW)**	**c/s Area Total (mm^2)**
A	Smith et al. [9], [10], [20]	1998	restoration of paralyzed muscle, MES	RF-ind.	2	100	12	FSM	1	34.1333	96.00	937.50
B	Eggers et al. [21], [22], [23]	2000	ICP-based diagnosis for brain diseases	RF-ind.	1	100	10	no	0.125	81.9200	0.24	58.50
C	Rollins et al. [24]	2000	continuous ECG for spontaneous cardiac arrhythmias	battery (ext.)	8	1000	12	FSM	2	0.8533	34.00	4209.67
D	Valdastri et al. [11]	2004	gastric-pressure monitoring	battery	1	25000	10	8-bit μC	4	0.3277	50.40	162.00
E	Au-Yeung et al. [25]	2004	continuous AEG, delivery of atrial ATP	battery	4	333	10	8-bit μC	8	6.1502	115.30	5106.00
F	Liang et al. [26]	2005	ENG	RF-ind.	1	11000	10	8-bit μC	n/a	0.7447	90.00	1350.00

IV. IMPLANT STUDY CASES

For selecting representative study cases of the implant application domain, we draw upon the extensive survey performed by Strydis et al. [1] who has investigated more than 60 cases of experimental as well as commercialized implantable devices. The selected applications will help provide diverse operational requirements for our targeted SiMS processor(s).

In order for a direct comparison (and fitting) with the candidate SiMS processor(s) to be made possible, we have to place the study cases in the same design space as the one traversed by ImpEDE. That is, we need to know the worst-case execution time, the power consumption and the area cost of each of the studied implantable systems. This requirement limits the number of eligible systems to only 6, as shown in Table IV, yet the scope of applications addressed is diverse.

As illustrated, actual implant-chipset sizes have been employed for the area metric. By 'chipset' are implied the dimensions of any design and assembly type ranging from fully integrated and multi-chip module (MCM) to PCB-mounted. Figures were also available for the implant chip-only size (in mm^2) - e.g. processor die but no supporting PCB - and for the implant package size (in mm^3). However, the chipset area was finally preferred so as to allow more direct and fair comparisons with the XTREM processor plus off-chip memory. Needless to say, we have insisted in including the memory in this study as the initial analysis (see section I) revealed rising trends in memory usage for future implants.

As far as power consumption is concerned, the most frequently reported figure in the actual implantable systems is active (peak) power which was the power measured during full load. This is the power simulated by XTREM as well, since XTREM does not support any low-power or sleep modes of operation, and it has been used here as the power-consumption metric.

Last, for the performance metric, some estimations had to be made to bring the real and simulated systems at the same level of comparison. All case studies are devices with periodic monitoring windows, thus exhibit a specific sampling rate, as shown in Table IV. This rate (or frequency) signifies (when inverted) the maximal amount of time they have to read a sensory sample and process it before the next sample arrives. In effect, this is the worst-case execution time of the implant. Note that we might have used the 'Core frequency' as a measure of the processing rate but this would be accurate only for designs with very simplistic cores as #A, #B and #C. For the rest of the cases whereby a full μC is used, the core frequency is much higher (typically three orders of magnitude) than the actual sampling frequency and, thus, does not reflect the real-time deadlines of the implant.

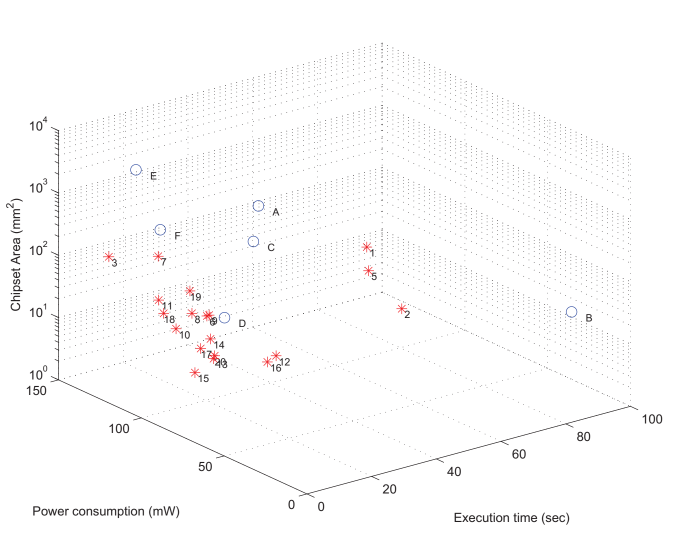
However, the performance metric for the study cases (as the inverse of the sampling rate) is not yet completely normalized with respect to that of our processor configurations. The reason is that, as discussed in the previous section, we have made our processor configurations consume EMG input data of $10\ KB$. The study cases, on the other hand, are assigned (by design) the task of consuming a single sample of size equal to the ADC resolution used (e.g. $8\ bits$), from each sensor they have on-board. Therefore, for each study case to collect $10\ KB$ of sample data, a longer execution time is needed which is inversely proportional to the number of available sensors. The normalized, worst-case execution time is then given by:

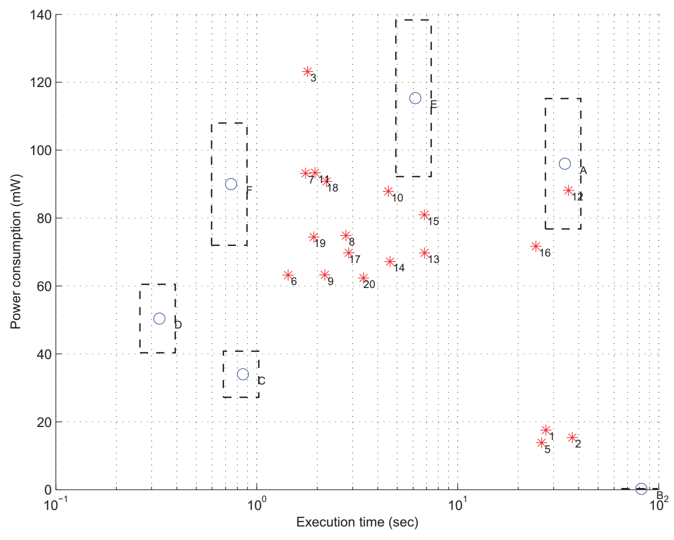
$$ET_{norm} = \frac{10\ KByte}{F \times N \times S}, \qquad (1)$$

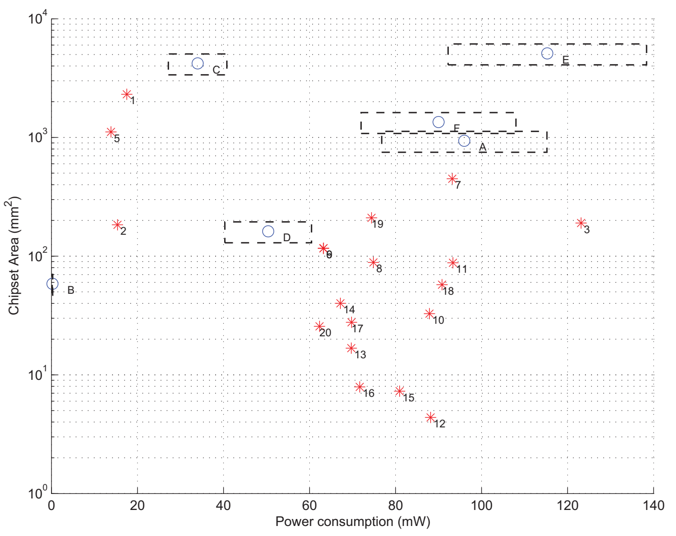
where F is the sampling frequency (in Hz) of the sensor(s), N is the ADC resolution (in bits) and S is the number of working sensors on the implant. This is the execution time given in Table IV. It should be noted that formula (1) does not account for any further processing of the data once acquired. On the contrary, our evolved processor configurations perform significant processing tasks, as discussed in section III-B. This has been done to promote our effort in designing implant processors under worst-case constraints.

V. EXPLORATION RESULTS

Since we have investigated 3 design metrics, the design space is a cube, as shown in Fig. 4a. In this Figure, our


(a) 3D view


(b) Worst-case, single-loop execution time - Average power consumption


(c) Average power consumption - Total chipset area

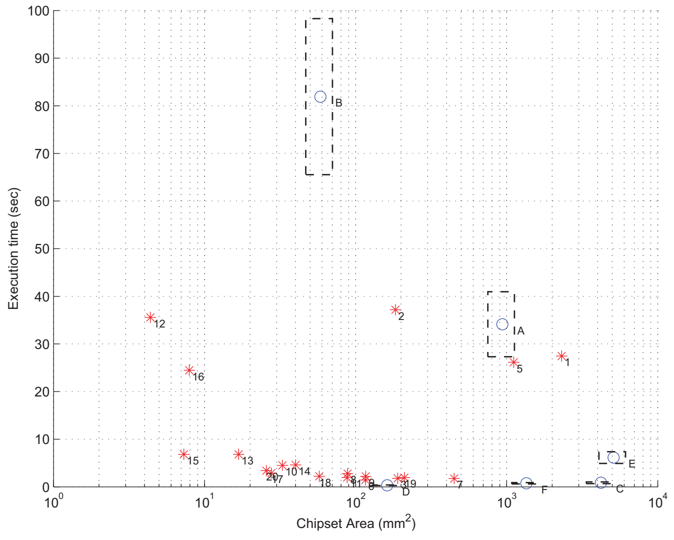

(d) Total chipset area - Worst-case, single-loop execution time

Fig. 4. Comparison of study cases and DSE results for 10 KB workloads running on the selected benchmarks.

processor design points (denoted with numbers) and the case-study points (denoted with letters) have been plotted. For clarity purposes, 2D Figs. 4b, 4c and 4d have also been plotted. The bounding boxes around the study cases represent 10% confidence intervals to compensate for the uncertainty introduced when trying to fit the study cases in the design space. Coupled with our worst-case design constraints, these intervals guarantee an extra measure against errors.

From the Figures, we notice that implant devices #A and #E are dominated by most of our processor points in all three dimensions. Therefore, we can easily replace these applications (restoration of paralyzed muscle, atrial electrogram, anti-tachycardia pacing) by any one of our processor configurations.

Device #B, on the other hand, is largely dominated in terms of execution time and area but not in terms of power. It, in fact, is the point in the 3D space with the lowest power profile ($0.24\ mW$) across processor configurations and study cases alike. It is, therefore, an outlying, marginal implant case in terms of power. Looking at the application details [21], we see that, indeed, #B is a minimal device measuring intra-cranial pressure and consuming such low power that it is connected and sufficiently powered by an external power source (i.e. power is RF-induced transdermally). Therefore, power is not an issue for this application and any of the processors that dominate it across the other two dimensions may replace it. Although all configurations are mutually non-dominated, likely candidates could be configurations #20, #14 and #17.

Furthermore, we see that devices #C, #D and #F are both overdesigned for performance. Of those, devices #C and #F respectively perform continuous-ECG and ENG monitoring, which are both quite demanding in terms of throughput (as indicated also by their reported sampling rates). We observe, however, that both devices behave significantly worse than most processor configurations in terms of area and worse than about half in terms of power. Given that these applications are high-performance ones (in the context of biomedical implants), if we would like to address them with our envisioned SiMS processor, we could trade the observed area and power margin for an extra *hardware accelerator* capable of delivering the extra performance required. Good candidates for such an accelerated processor could be configurations #5, #20 and #6.

Device #D, on the other hand, performs gastric-pressure monitoring at an extremely high sampling rate of $25\ kHz$ (the highest across all study cases). Nevertheless, gastric pressure does not change that rapidly, therefore, in our opinion processor points with much higher execution times (the original $\times 10$, i.e. around $3 - 4\ sec$) can also safely cover the application of device #D. Processor configurations #17, #20 and #14 can make this time deadline and, given the fact that they also exhibit better power and area metrics than #D, they could be considered as good solutions for this application.

With the above analysis in mind, we can see that the minimalistic processor configuration #20 is one of the most promising ones across all studied devices. It is interesting to mention that its characteristics (no sophisticated BPRED scheme, both L1 caches present with I$ larger than D$ etc.) agree with the partial findings of a previous study on BPRED and cache scheme for biomedical implants [27]. Of course, this is not to say that the rest of the suggested configurations from the above analysis are not interesting solutions. Keep in mind that all considered configurations are non-dominated, Pareto-optimal solutions of the design space. In fact, in the future we tend towards a highly structured, systematic, implant-design approach in which a small family of processor configurations is designed and is able to cover a large part of (if not all) the implant application field.

VI. Conclusions

In this paper, we have presented a complete approach towards the systematic, educated and automated microarchitectural specification of processors for biomedical, microelectronic implants. We have provided 19 Pareto-optimal processor alternatives investigating a large set of hardware parameters such as I-cache and D-cache geometries, branch-prediction policy and memory latency. To the best of our knowledge, we have also provided the first comparison between the suggested processor configurations and existing, documented implantable devices across a wide range of applications. To manage this, we have established a means of direct comparison based on careful assumptions that take into account the unavoidable inaccuracies of our tools. In so doing, we have proposed processors that can operate under worst-case conditions, i.e. they are suitably provisioned for the mission-critical implant applications.

Even so, as future work, we wish to expand our DSE framework to also optimize for system reliability. To do so we need to introduce a fourth metric (one currently considered is fault coverage) and expand our tools accordingly. Furthermore, we are already busy porting XEEMU to our system as a more error-free and accurate replacement for XTREM.

VII. Acknowledgements

This work has been partially supported by the ICT Delft Research Centre (DRC-ICT) of the Delft University of Technology. Many thanks are due to Erik de Vries and Eef Hartman for their excellent technical support, primarily in setting up and operating the DSE computer cluster. Without them, this work would not have been possible.

References

[1] C. Strydis *et al.*, “Implantable microelectronic devices: A comprehensive review,” Computer Engineering, TU Delft,” CE-TR-2006-01, Dec. 2006.

[2] R. Sanders and M. Lee, “Implantable pacemakers,” in *Proceedings of the IEEE*, vol. 84, Mar. 1996, pp. 480–486.

[3] F. Nebeker, “Golden accomplishments in biomedical engineering,” in *IEEE Engineering in Medicine and Biology Magazine*, vol. 21, Piscataway, NJ, USA, May - June 2002, pp. 17–47.

[4] C. Strydis and G. Gaydadjiev, “The Case for a Generic Implant Processor,” in *30th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC’08)*, August 2008, pp. 3186–3191.

[5] “Smart implantable Medical Systems,” http://sims.et.tudelft.nl.

[6] K. Fernald, T. Cook, T. M. III, and J. Paulos, “A Microprocessor-Based Implantable Telemetry System,” in *IEEE Computer*, vol. 24, Mar. 1991, pp. 23–30.

[7] K. Fernald, B. Stackhouse, J. Paulos, and T. Miller, “A System Architecture for Intelligent Implantable Biotelemetry Instruments,” in *Proceedings of the Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBS)*, vol. 11, Nov. 1989, pp. 1411–1412.

[8] T. M. et al., “The Hector Microprocessor,” in *Proceedings of the IEEE International Conference on Computer Design (ICCD)*, 1986, pp. 406–411.

[9] B. Smith, Z. Tang, M. Johnson, S. Pourmehdi, M. Gazdik, J. Buckett, and P. Peckham, “An externally powered, multichannel, implantable stimulator-telemeter for control of paralyzed muscle,” in *IEEE Transactions on Biomedical Engineering*, vol. 45, 1998, pp. 463–475.

[10] S. Pourmehdi, P. Strojnik, P. Peckham, J. Buckett, and B. Smith, “A custom-designed chip to control an implantable stimulator and telemetry system for control of paralyzed muscles,” in *Artificial Organs*, vol. 23, May 1999, pp. 396–398.

[11] P. Valdastri, A. Menciassi, A. Arena, C. Caccamo, and P. Dario, “An implantable telemetry platform system for in vivo monitoring of physiological parameters,” in *IEEE Transactions on Information Technology in Biomedicine*, vol. 8, Sept. 2004, pp. 271–278.

[12] S. Salmons, G. Gunning, I. Taylor, S. Grainger, D. Hitchings, J. Blackhurst, and J. Jarvis, “ASIC or PIC ? Implantable stimulators based on semi-custom CMOS technology or low-power microcontroller architecture,” in *Medical Engineering & Physics*, vol. 23, 2001, pp. 37–43.

[13] I. T. R. for Semiconductors (ITRS), “[online] available: http://www.itrs.net/common/2004update/2004update.htm,” 2004.

[14] D. Dave, C. Strydis, and G. N. Gaydadjiev, “ImpEDE: A Multidimensional Design-Space Exploration Framework for Biomedical-Implant Processors,” in *To appear in: Proceedings of the 21th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP’10)*, July 7-9 2010.

[15] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, “A Fast Elitist Multi-Objective Genetic Algorithm: NSGA-II,” *IEEE Transactions on Evolutionary Computation*, vol. 6, pp. 182–197, 2000.

[16] G. Contreras *et al.*, “XTREM: A Power Simulator for the Intel XScale Core,” in *LCTES’04*, 2004, pp. 115–125.

[17] D. Dave, “Automated implant-processor design: An evolutionary multiobjective exploration framework,” Master’s thesis, TU Delft, 2010.

[18] Z. Herczeg, A. Kiss, D. Schmidt, N. Wehn, and T. Gyimóthy, “Xeemu: An improved xscale power simulator,” *Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation*, pp. 300–309, 2007.

[19] C. Strydis, D.Dave, and G. Gaydadjiev, “ImpBench revisited: An extended characterization of implant-processor benchmarks,” in *Submitted to: International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS’10)*, Samos, Greece, 2010.

[20] S. Pourmehdi, P. Strojnik, P. Peckham, J. Buckett, and B. Smith, “A custom-designed chip to control an implantable stimulator and telemetry system for control of paralyzed muscles,” in *Proceedings of the 6th Vienna International Workshop on Functional Electrical Stimulation*, Vienna, Austria, 2224 September 1998.

[21] T. Eggers, C. Marschner, U. Marschner, B. Clasbrummel, R. Laur, and J. Binder, “Advanced hybrid integrated low-power telemetric pressure monitoringsystem for biomedical applications,” in *MEMS’00*, 2000, pp. 329–334.

[22] ——, “Advanced hybrid integrated low-power telemetric pressure monitoring system for biomedical applications,” in *IEEE Proceedings of Microelectromechanical Systems (MEMS)*, Miyuzaki, Japan, 2000, pp. 329–334.

[23] K. Hille, J. Draeger, T. Eggers, and P. Stegmaier, “[Technical construction, calibration and results with a new intraocular pressure sensor with telemetric transmission] [Article in German],” in *Klinische Monatsblatter fur Augenheilkunde*, vol. 218, May 2001, pp. 376–380.

[24] D. Rollins, C. Killingsworth, G. Walcott, R. Justice, R. Ideker, and W. Smith, “A telemetry system for the study of spontaneous cardiac arrhythmias,” in *IEEE Transactions on Biomedical Engineering*, vol. 47, July 2000, pp. 887–892.

[25] K. Au-Yeung, C. Johnson, and P. Wolf, “A novel implantable cardiac telemetry system for studying atrial fibrillation,” in *Physiological Measurement*, 11 August 2004, pp. 1223–1238.

[26] C. Liang, J. Chen, C. Chung, C. Cheng, and C. Wang, “An implantable bi-directional wireless transmission system for transcutaneous biological signal recording,” in *Physiological Measurement*, vol. 26, Feb. 2005, pp. 83–97.

[27] C. Strydis and G. N. Gaydadjiev, “Evaluating Various Branch-Prediction Schemes for Biomedical-Implant Processors,” in *Proceedings of the 20th IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP’09)*, July 2009, pp. 169–176.

Curriculum Vitae

Dhara Dave