Processor Architecture and Data Buffering

Hans Mulder, *Member, IEEE*, and Michael J. Flynn, *Fellow, IEEE*

Abstract—**The tradeoff between visualizing or hiding the highest levels of the memory hierarchy impacts both performance and scalability. We discuss these tradeoffs by comparing a set of architectures from three major architecture families: stack, register, and memory-to-memory. The stack architecture is used as reference. We show that scalable architectures require at least 32 words of local memory and therefore are not applicable for low-density technologies. We also show that software support can bridge the performance gap between scalable and nonscalable architectures. A register architecture with 32 words of local storage allocated interprocedurally outperforms scalable architectures with equal sized local memories and even some with larger sized local memories. When in addition to quality compile-time support, a small cache (e.g., $\geq$ 64 words) is added to an unscalable architecture the performance advantage (cycles) of unscalable architectures becomes significant. For example, a 32-register architecture with a 512 byte cache executes 20% less cycles when compared with a 8-set multiple overlapping set organization.**

Index Terms—**Caches, computer architecture, register allocation, register set, register windows, stack buffers.**

I. Introduction

VISIBILITY and scalability are two opposite architecture features; the first aims at exhibition of low-level features for the sole reason of exploiting them, while the second aims at hiding these features to allow greater freedom in implementing the architecture.

The architecture feature targeted in this paper is local memory. A fixed single register set is a feature visible in **register** architectures that cannot be scaled with technology or cost/performance requirements, and that relies completely on exploitation during compile time. Stack buffers or stack caches [1]–[3], which offer potentially optimal use in buffered **memory-to-memory** architectures, are a feature invisible in the architecture. They are therefore scalable with technology and cost/performance requirements, and independent of compile-time software. Multiple overlapping register sets [4] are somewhere in between single register sets and stack-buffer organizations with respect to scalability and visibility; generally the size of the set and the overlap are architecturally defined, while the number of sets is an (architecturally invisible) implementation parameter.

The three architecture families studied are **stack**, **register** (either single or multiple sets), and buffered **memory-to-memory**. Fig. 1 illustrates their differences with respect to the instructions required to calculate a simple expression. **Stack** includes only an evaluation stack of sufficient size to buffer temporary results during evaluation of expressions. **register** includes a register set (*srs* or *mrs*) which can be used to buffer variables besides operating as an evaluation stack. Buffered **memory-to-memory** does not include an architecturally visible buffer, but employs a three-ported stack buffer (*stack*) architecturally transparent to main memory.

To illustrate the differences between these architecture models, Table I shows the expansion of an expression in four different instruction sequences. The first column shows **stack**, the second column shows **register** with the registers only holding temporaries, the third shows **register**, but now with variables assigned to registers, and the fourth shows buffered **memory-to-memory**. Note that the differences are solely related to local memory and operand access; for the rest, all architectures are assumed to have identical instruction vocabularies.

A. Performance Measures

To compare the local memory characteristics of different architectures the most obvious measure is the data traffic. The data traffic is the remaining traffic to the main memory system after (optional) buffering. Traffic measurements are all presented as data-traffic ratio, or data traffic relative to the traffic of **stack** ($tr_{arch} = T_{arch}/T_{stack}$).

Data-traffic ratio, however, does not suffice as a performance measure when a closer correlation with real performance ($1/time$) is necessary. A closer correlation is necessary, when comparing architecture-and-buffer combinations that are structurally different; for example when comparing a stack buffer with a combination of a register set and a data cache. A measure which does correlate with real performance is the cycle ratio (cr) [5]. The cycle ratio reflects the number of cycles spent fetching and storing data for a particular architecture-buffer combination (C_{arch}) relative to these cycles for **stack** (C_{stack}). The cycle ratio assumes that the highest-level in the memory hierarchy (e.g. register set, evaluation stack) allows two reads and one write concurrently, while all lower levels are accessed sequentially. The cycle ratio is always presented as ($cr_{arch} = C_{arch}/C_{stack}$).

B. Methodology and Workload

One essential requirement when comparing architecture features is to eliminate the effect of extraneous features. This kind of *fair-basis* comparison is greatly simplified by means of the Computer Architect's Workbench [6], [7], a set of tools designed with the sole purpose of doing architecture

Manuscript received September 15, 1990; revised October 1, 1991. This work has been supported by NASA under Contract NAG2-248, using facilities provided under Contract NAGW 419.

H. Mulder is with the Department of Electrical Engineering, Delft University, The Netherlands.

M. J. Flynn is with the Department of Electrical Engineering, Stanford University, Stanford, CA 94305.

IEEE Log Number 9200205.

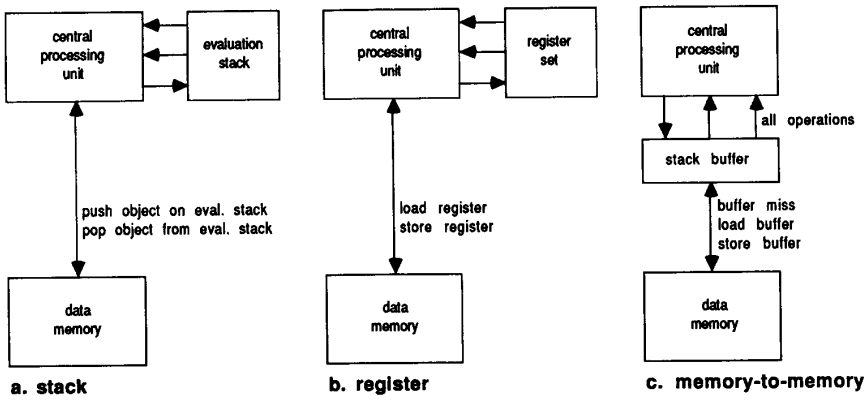


Fig. 1. Overview of the three architecture families.

TABLE I
ARCHITECTURE FAMILY VERSUS EXECUTED INSTRUCTIONS

A = B + (C * D)			
Stack	Register		Memory
	not allocated	allocated	
push C	load C,R_1	$R_{tmp} = R_C * R_D$	T = C * D
push D	load D,R_2	$R_A = R_B + R_{tmp}$	A = B + T
*	$R_2 = R_1 * R_2$		
push B	load B,R_1		
+	$R_1 = R_1 + R_2$		
pop A	store R_1,A		

research. These tools currently consist of compiler front ends for different languages, register allocators, and simulators for different architectures and memory hierarchies. The tools are currently being extended to include timing information and to improve ease of use.

The workload for this study consists of a set of five Pascal programs: *ccal*, an interactive calculator; *comp*, a file-compare program; *macro*, a macro expander; *pasm*, a P-code assembler; and *pcomp*, a recursive-descent Pascal compiler. These five benchmarks do a significant amount of input and output, but this I/O activity is not taken into account in the simulations. In other words, the I/O primitives are assumed not to interfere with the data buffering characteristics. The measurements presented include both the unweighted average and the individual results of some of the benchmarks. See Table II for the total number of calls, references, and instructions of the five benchmarks.

C. Paper Overview

The remainder of the paper consists of seven sections. Section II presents the **stack** architecture which will be used as reference for the other architecture families. Section III and IV present data on architectures with single fixed register sets and architectures with multiple overlapping register sets, respectively. Section V presents data on **memory-to-memory** architectures equipped with stack buffers. Section VI presents data on architectures with small register sets but equipped with a small cache. Section VII combines and compares the data on the three different families, followed by conclusions on the visibility-scalability question in Section VIII.

II. STACK ARCHITECTURE

The **stack** architecture is the base architecture model to which most measurements are related. While an unbuffered **memory-to-memory** architecture (with two- or three-address formats) could have been chosen as a base, **stack** was preferred since it eliminates references to temporaries within an expression. As shown later, such references account for almost 50% of data references and can be readily eliminated with just a few registers or an evaluation stack.

Local memory in **stack** consists solely of an evaluation stack which holds temporary results. **Stack** also includes three registers to hold state variables; one pointing to the base of the activation record of the currently executing procedure, one to the base of the activation record with global variables, and one to the top of the heap. Memory is referenced solely by means of push and pop instructions.

The **stack** run-time model consists of two stacks: one for the procedure activation records and one for the heap. A procedure activation record consists of local variables, parameters, and the run-time-management variables (return program counter, dynamic link, and static link).

An unbuffered three-address **memory-to-memory** architecture would make all (100%) references indicated in Fig. 2(a). In the **stack** architecture, the evaluation stack captures about 47% of these. The remaining 53% of references form a baseline data traffic (**stack** data traffic), as any architecture of interest will provide sufficient registers to capture this expression-evaluation traffic. Of the baseline references, 73% originates in the source program as references explicitly specified by the programmer, 3% are created by the compiler to support certain Pascal constructs, 17% manage the run-time stack, and 7% transfer multi (32-bit) word references over a single word bus [see Fig. 2(b)].

Within our model, a two element stack captures 85% of the evaluation stack references, but three elements are required statistically to capture all evaluation stack references. Table III shows the traffic ratio as a function of the number of

TABLE II
NUMBER OF PROCEDURE CALLS, REFERENCES, AND INSTRUCTIONS FOR THE FIVE BENCHMARKS (**stack** ARCHITECTURE)

	ccal	comp	macro	pasm	pcomp	total
calls	16 676	10 508	9666	15 172	55 868	107 890
references	582 389	4 300 546	412 530	2 575 369	3 076 829	10 947 663
instructions	1 133 765	9 281 305	633 056	4 548 038	5 683 284	21 279 448

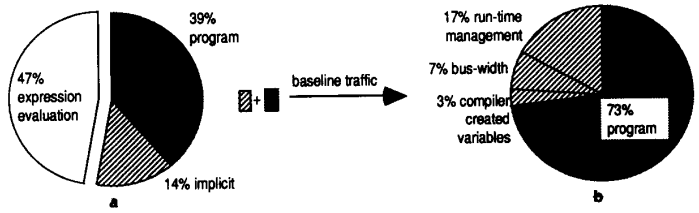


Fig. 2 Reference taxonomy.

TABLE III
TRAFFIC FOR **stack** AS A FUNCTION OF THE NUMBER OF EVALUATION STACK ELEMENTS

#	ccal	comp	macro	pasm	pcomp	avg
0	1.86	2.27	1.76	1.81	1.71	1.88
2	1.12	1.37	1.06	1.13	1.07	1.15
≥ 3	1	1	1	1	1	1

stack entries relative to the **stack** architecture with sufficient evaluation stack entries. The first row shows the relative traffic for an architecture without an evaluation stack.

By mapping the evaluation stack elements of **stack** in explicitly addressed registers together with the three state registers (local base, global base, and heap pointer), a six-register architecture can be created which is identical to **stack** with respect to data traffic and instruction count (see also Table I). Additional registers, however, may be used to hold variables and to capture references directed to these variables. The following two sections present the effects on the data traffic when using these additional registers.

III. SINGLE-REGISTER-SET ARCHITECTURES

Single register sets (*srs*'s) are an integral part of the instruction set architecture (ISA). Because registers are part of the ISA, the compiler or register allocator has to exploit them. Thus, the way the compiler uses the registers affects performance significantly. Single-register-set organizations are depicted by *srs(size,allocator,run-time)* in the following discussion. The first parameter, *size*, is an architecture parameter and indicates the number of registers in the set. The choice of allocators (second parameter) for this paper is limited to the global register allocator of the Stanford U-code system *uopt* [8] and an interprocedural allocator, *inter* [5]. The third parameter indicates the allocation of the run-time-management variables.

A. Register Allocation

Uopt operates on a procedural scope, uses a *priority based coloring* algorithm [9], [10] and a *caller save* strategy for saving/restoring registers across procedure calls.

Inter builds the callgraph and symboltable of the whole program and allocates registers for the entire lifetime of the program in a depth-first call order. Within a procedure, *inter* employs a simple reference-counting strategy. Since mutually exclusive procedures can share private registers, the total number of registers required is not proportional to the number of procedures but on the average to the *log* of this number. Recursive procedures are only allocated globally; their registers are saved to memory on subsequent calls (in the recursive path). Interprocedural register allocation by traversing the call graph depth first has been described before by Wall [11] for link-time allocation, Steenkiste [12] for Lisp, and Chow [13] for the MIPS compiler.

B. Run-time Model

register (either *srs* or *mrs*) maintains the same run-time model as **stack**. Every variable is still allocated to a slot in an activation record, but the variable may be allocated to and used from a register. The layout of the activation records is the same as in **stack** except when run-time variables are taken into account during allocation.

Because *Uopt* operates on U-code which has an implied run-time system, *Uopt* never takes run-time variables into account. Allocation of run-time variables when using *uopt* simply implies the use of a register-based display (when sufficient registers are available) instead of a memory-based static link. These two options are indicated as *mem-sl* and *reg-disp* in the following figures.

Since most run-time-management variables (program counter and dynamic link) are private to a particular procedure, *inter* is able to allocate these variables to registers. When a *caller* contains just register-mapped variables, *inter* removes the dynamic link of the *callee*. *Inter* removes all static links in nonrecursive procedures and substitutes the appropriate register identifiers or memory addresses for nonlocal variables. Nonlocal access into recursive procedures occurs still through a memory-based static link.[1]

[1] Static linkage is only necessary from one procedure to another one in the same recursive complex.

TABLE IV
REFERENCE FREQUENCY AS A FUNCTION OF VARIABLE TYPE

segment	ccal	comp	macro	pasm	pcomp	avg.
local	0.44	0.68	0.44	0.39	0.29	0.45
global	0.10	0.02	0.16	0.17	0.16	0.12
run-time	0.20	0.07	0.18	0.32	0.12	0.18
subtotal	0.74	0.77	0.78	0.87	0.57	0.75
rest	0.26	0.23	0.22	0.13	0.43	0.25

C. Allocation Effectiveness

Typically the register allocator targets simple stack variables (local and global), and, if possible, also run-time-management variables for allocation. Table IV shows the maximum achievable gain of three allocation targets. If only simple local variables are allocated on the average a traffic ratio reduction of 0.45 is possible. Global variables may yield an additional 0.12 and run-time variables another 0.18. Eliminating all references to the run-time stack results in an average traffic ratio of 0.25, leaving only references to structured and heap variables uncaptured.

In assessing register-allocation effectiveness, we wish to exclude those registers that are unavailable to the allocator. In presenting our results we assume that six registers are required for dedicated purposes and are unavailable to the allocator. A possible layout for these six registers is: local-base pointer, global-base pointer, heap pointer, two evaluation-stack elements, and a register containing immediates (e.g., zero). Reserving these registers allows us to present results based on the expected overall size of the register set. Thus, if the instruction set specifies 16 registers, ten are available for allocation.

By using global allocation a significant reduction in traffic can be obtained. Fig. 3 illustrates this reduction; an eight-register architecture (implying just two registers for the allocator) has an average traffic ratio of 0.70. Doubling the register set to 16 yields an average ratio of 0.65, and leaves sufficient registers to implement lexical (or static) scoping by means of a register based display to reduce the traffic ratio to 0.55.

Using *inter* instead of *uopt* shows a more significant traffic reduction, and the advantage of using larger register sets. When the allocator ignores run-time variables, the average traffic ratio runs from 0.61 for a 16-register set to 0.51 for 32 registers and 0.48 for 128 or more registers. Including optimizing the use of run-time-management variables and allocating return program counters and dynamic links were possible and necessary yields an average ratio of 0.49 for 16 registers, 0.4 for 32 registers, and 0.33 for 128 registers. Note that *inter* has a complete overview of the register requirements of the whole program it is less constrained by the 6-register overhead. *Inter* generally can find an available register for state or evaluation stack registers when actually needed instead of permanently allocating one.

Interprocedural register allocation requires few registers to perform very well. Elimination of most of the simple local references, however, requires 32 registers on the average, and buffering of run-time variables requires an additional 32 registers. Interprocedural register allocation, however, is not universally applicable because it requires a compile-time state where the register demand for the entire program is known. For separately compiled modules this would be in the linking phase; incrementally compiled Lisp, Prolog or Small Talk systems require a dynamic allocation scheme.

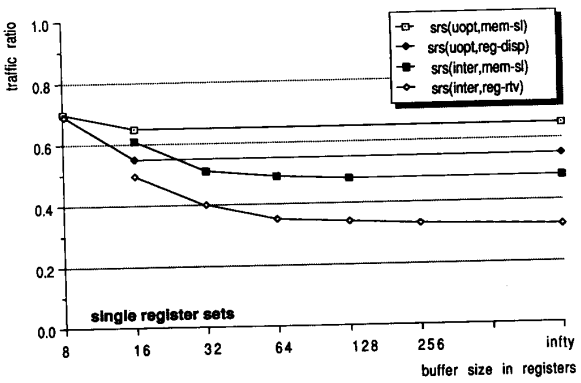


Fig. 3. Traffic ratio for register architectures allocated by means of *uopt* and *inter*.

Because interprocedural allocation is inexpensive with respect to compile time ($O(calls)$), even application on a small scale (e.g., within a single module) should be considered.

IV. MULTIPLE REGISTER SET ARCHITECTURES

Multiple register set (*mrs*) architectures reduce the register save and restore penalties inherent to single register set architectures by providing each new procedure with a private register set [4], [14]. The set change is accomplished automatically on call and return. The save and restore penalty is now replaced by a (smaller) over- and underflow penalty, incurred when insufficient sets are available to accommodate the whole call chain of the executing program.

By means of an aliasing detection-and-resolution mechanism [14] in hardware it is possible to access variables which are not in a register of the set used by the current procedure, but do reside in a register not yet overflowed to memory. Such a mechanism requires the procedure activation records to have memory equivalents when they are mapped into registers. Whenever a memory address points into a register mapped section of the stack of activation records, the data from the register need to be used. Because Pascal programs access nonlocal variables[2] and the hardware is relatively simple, the following discussion and figures always assumes the inclusion of such a hardware mechanism in the presented *mrs* organizations.

In *mrs* organizations architecture visibility is limited to 1) the registers in a single set, 2) the knowledge that new registers will be provided upon procedure entry or after an explicit set request, and 3) the memory transparency provided by an aliasing mechanism. Consequently, the complexity of

[2]For the five benchmarks, on the average 5% of all references to the run-time stack were made through the var parameter and nonlocal access mechanism. Note though that this percentage is strongly language and application dependent.

register allocation can be reduced to a primitive form of global allocation, and the architecture becomes scalable. More chip or board area dedicated to the *mrs* will hold more sets, and may therefore reduce traffic and may increase performance.

In the following figures, *mrs* organizations will be depicted by *mrs(overlap, locals,allocation,sets)*, where *overlap* indicates the set overlap in registers, *locals* the number of nonoverlapping registers, *allocation* whether explicit register allocation is performed (*no*, *global*, or *inter*), and *sets* optionally indicates the number of sets in the buffer (Fig. 4). For example, in *mrs(8,8)* sixteen registers are added per window, but the processor addresses 24 registers (8 input parameters, 8 locals, 8 output parameters) and perhaps 8 global registers in each environment.

A. Allocation Strategies for mrs Organizations

Mrs architectures generate two kinds of references which are related to the operation of the *mrs*. The first concerns references caused by nonoptimal set organizations. Allocation on a procedure basis (global allocation) can reduce these references. The second concerns the under- and overflow references caused by an insufficient number of sets in the buffer. These references may be reduced by using interprocedural allocation, which causes multiple procedure contexts to reside in the same set. The following three paragraphs discuss the different allocation strategies.

No allocation: Besides references caused by *mrs* under- and overflows, some references potentially captured by the *mrs* are directed to memory because every set has a limited size, forcing variables to reside in memory instead of registers. Overlapping register sets have an additional constraint, which limits the number of parameters passed without making use of a run-time stack in memory. As shown in Halbert [15], an overlap size of 8 and a local size of 8 registers, or 24 visible registers in every window, holds nearly all parameters and local variables of a set of C programs. An *mrs* with this organization may hold the variables of a program so comfortably that the instance of procedures requiring more registers than are available is infrequent; those variables can be mapped into registers according to the programmer's declaration order. Fig. 5 shows traffic ratios of two organizations, *mrs(4,4,no,∞)*, and *mrs(8,8,no,∞)*, allocated according to the declaration order in the source program.[3] Processors based on RISC *mrs* [4] have been introduced by Sun Microsystems under the SPARC label. In our terminology these are *mrs(8,8)*. The total number of sets is an implementation parameter, but usually 128 registers are implemented. SPARC implementations do not include the alias detection hardware referred to earlier and hence, for our benchmarks, would have performance somewhat less than the *mrs(8,8)* presented here.

Global allocation: Instead of allocation according to declaration order, the compiler can also perform more intelligent register allocation to increase the buffer utilization. The allocation strategy can be very simple, for example by mapping variables into registers according to their expected usage and for the lifetime of the procedure. It can also be more complex, for example by analyzing the dataflow and using coloring algorithms for the actual allocation [9], [10]. Fig. 5 shows the traffic ratio of *mrs(4,4,global,∞)* and *mrs(8,8,global,∞)* with registers allocated according to the simple allocation scheme. Allocation makes a significant difference for the organizations with small sets (*mrs(4,4,no)* versus *mrs(4,4,global)*); a quantification of the improvement is not very useful because it depends on the (arbitrary) performance of the *mrs*'s without allocation.

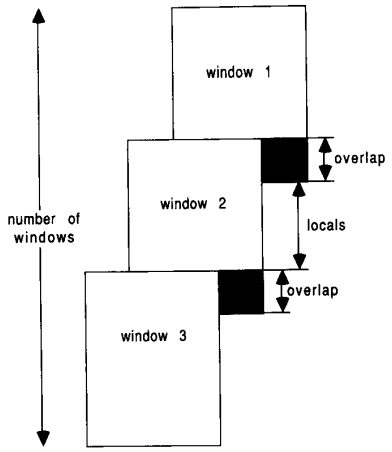


Fig. 4. *mrs(overlap, locals, allocation, number of windows).*

Interprocedural allocation: Interprocedural allocation as applied on single register sets in Section III can also be applied to *mrs* organizations. As a result, a procedure call will not necessarily cause the sets to change. The interprocedural allocator will place as many procedure contexts in the same set as possible. There are several possible strategies to place multiple contexts in a single set. The simplest strategy adds a context to a set if and only if the whole context fits. If only a partial fit is possible a new set must be used. This strategy causes a reduction in under and overflow traffic and has identical hit ratios as global allocation only. For example the column *mrs(8,8,global)* in Fig. 5 is also the column *mrs(8,8,inter)*. The advantage of this strategy is that it is independent of the number of sets available. A more complex strategy would consider leaving variables in memory to obtain a better fit or even changing the register file in the middle of a procedure instead of on procedure boundaries. Particularly the former strategy requires knowledge on the number of sets available. For this paper we only show the effects of applying the simple strategy to allocation for *mrs(8,8)*.

Fig. 6 shows the effect of interprocedural allocation on the under- and overflow traffic of *mrs(8,8,inter)*. Although global and interprocedural allocation are independent, we assume that the use of interprocedural allocation implies the use of some form of global allocation (reference counting for *inter*). Inclusion of *mrs(8,8,no)* in Fig. 6 illustrates this independence of global and interprocedural allocation (for our interprocedural strategy).

[3] Note that these ratios are fairly arbitrary, because the programmer can declare the variables in any order, yielding traffic ratios somewhere between best and worst case possible.

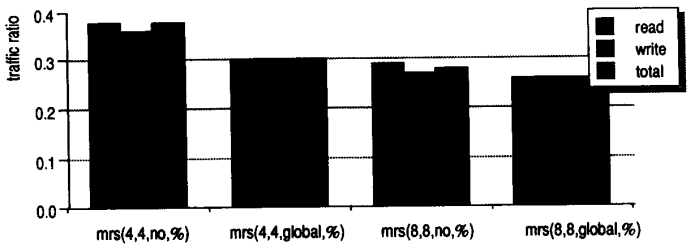


Fig. 5. Effect of register allocation on *mrs* performance.

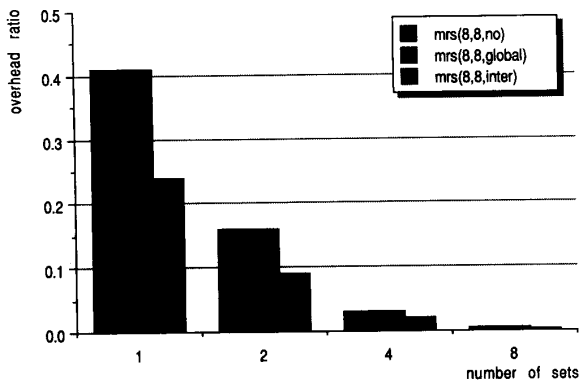


Fig. 6. Effect of interprocedural allocation on under- and overflow traffic (relative to baseline traffic).

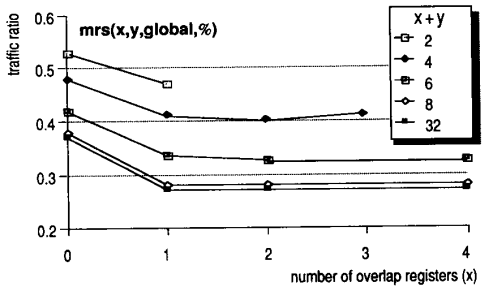


Fig. 7. Set size and overlap versus traffic ratio.

TABLE V
OVERLAP, LOCAL SIZE, AND GLOBAL REGISTERS FOR DIFFERENT *mrs* ORGANIZATIONS

Organization	overlap	locals	globals		
			4-bit[a]	5-bit[a]	6-bit[a]
mrs(8,0)	8	0	0	16	48
mrs(4,4)	4	4	4	20	52
mrs(16,0)	16	0	n/a	0	32
mrs(8,8)	8	8	n/a	8	40

[a]number of bits in register specifier (id bits)

B. Register Utility Versus Set Size

Three main issues determine the set organization within a *mrs* architecture: the effect of the organization on the traffic ratio, the register access time, and the effect on the number of available global registers.

Traffic ratio: To find the right division Fig. 7 plots the traffic ratio for five organizations with different set sizes (*overlap* + *local*) as a function of the overlap in registers. For the five benchmarks, using a set size of more than eight registers yields only a marginal improvement in traffic ratio. Note that locals may be allocated in the incoming parameter slots; this makes the performance only partially dependent on the overlap size. It appears that the Pascal programs in question require only a very limited overlap (one to three registers).

Intuitively increasing the number of overlap registers for a given set size should always reduce traffic ratio. The graph for a set size of 4 however shows an increase for an overlap larger than two and a set size of eight shows a constant ratio for overlaps larger than one. The allocator is the cause of the first effect by preferring parameters over locals for overlap registers without consideration for the performance effects. As a result a parameter may be allocated an overlap register, while a more frequently used local is denied a local register. The second effect implies that many parameters can be treated as locals[4] with unnoticeable performance effects (< 0.01).

Access time: The register-access time as a function of organization is mainly dependent on the ease of determining the register to be an incoming parameter, outgoing parameter, local register, or global register. The most advantageous organization with respect to access time is *mrs(8,8)* or *mrs(4,4)* because decoding is only a matter of checking the upper one or two bits of the register specifier.

Global registers: The amount of overlap also influences the number of global registers when all register-specifier bits are used (Table V). The number of global registers is $N_{global} = 2^{idbits} - 2 \times N_{overlap} - N_{local}$. Fig. 7 shows that the performance depends mainly on the total of overlap and local registers. When utilization of the global registers is feasible, the overlap should be as small as possible under the traffic ratio and register access time constraints. A particular use of the global registers in an *mrs* is holding the overhead registers described for *srs* organizations. Note however that *mrs* organizations require fewer overhead registers because most of them are mapped in the local registers.

C. MRS Architecture Performance and Comparison

Efficient buffering using multiple register sets requires a simple hardware organization together with at least global register allocation. The basic idea is to implement a hardware scheme with a small set size and set overlap, and cover these hardware limitations with global register allocation. Such

[4]Which means passing through memory before loading into a local register.

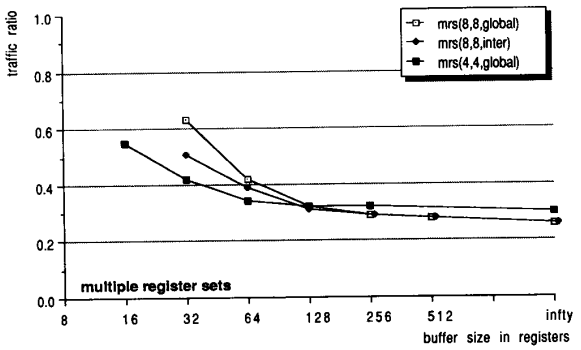


Fig. 8. Traffic ratio for multiple register-set architectures.

a scheme neither requires variable set overlap, nor variable set size, nor large low-utility sets. Optimally the complexity reduction in the organization allows the implementation of more sets.

Extending the basic MRS idea does not involve more hardware, but improved register allocation. Combining global with interprocedural allocation yields MRS organizations which are relatively independent of total set size and set overlap.

Fig. 8 presents the traffic ratio of *mrs(4,4,global)*, *mrs(8,8,global)*, and *mrs(8,8,inter)* as a function of the number of registers in the multiple sets. All organizations require at least four sets (*mrs(8,8,global,4)*, *mrs(8,8,global,4)*, or *mrs(8,8,inter,4)*) to perform acceptably ($tr \leq 0.5$). When sufficient sets are available, *mrs(4,4,global,∞)* generates on the average 15% more traffic than *mrs(8,8,global,∞)* or *mrs(8,8,inter,∞)*. Note that *mrs* organizations are only plotted for two or more sets. Building a single set *mrs*, although feasible, does not make much sense.

If technological independence is important, or, in other words, if performance should be acceptable over a wide range of buffer sizes, then either small sets ($\leq$ 8 registers) with additional global[5] allocation are essential. If, in addition, maximization of the global register set size is important then the overlap should be kept small (2–3 registers).

V. Buffered Memory-To-Memory Architectures

The third architecture family, **memory-to-memory**, does not includes a memory hierarchy on the architecture level, like **stack** (the evaluation stack) or **register** (the register set). Architecturally the memory system is completely *flat*. Similar to **register** however, **memory-to-memory** is able to explicitly address two source and one destination operand every instruction, and therefore **memory-to-memory** is able to exploit local memory as defined in Section I.

A. Buffer and Run-time Organization

A buffer for **memory-to-memory** is a local (noncache) memory that is not part of the instruction-set architecture. Because we are mainly interested in easily captured references, buffering is limited to the run-time stack. Because the run-time stack is the target of the local memory this kind of buffer is called a stack buffer (also called a stack cache [1] or contour buffer [3]). A stack buffer is comparable with an *mrs* by automatically providing new registers upon procedure entrance. Organizationally, the main difference with *mrs* is that the stack buffer provides variable overlap and variable local size. The stack buffer covers the top of the run-time stack and moves back and forth with the movements of this stack.

We distinguish three different organizations (Fig. 9) to buffer the run-time stack:

1) *stack(single)*, the buffered stack containing the global structure, global simples, local structures, and local simple variables.
2) *stack(single,global)*, the buffered stack containing global simples, local structures, and local simple variables.
3) *stack(split)*, a dual-stack organization where the buffer stack holds just the global simple and local simple variables, and a second nonbuffered stack which contains the global structure and the local structure variables.

These three organizations represent tradeoffs between complexity of stack management versus performance; *stack(split)* is the most complex organization, but offers the best performance.

The Bell-labs CRISP processor [2] is an implementation of a typical **memory-to-memory** architecture with a stack buffer of 32-words (*stack(single,global,32)*). Note, however, that the distinctions between *split, single,* and *global* are determined by software rather than hardware. Thus, the CRISP could use any strategy with suitable software support.

B. Run-time Organization Versus Performance

The choice of run-time organization is not arbitrary (Fig. 10). Including the global data in the buffer *stack(single)* causes anomalously bad performance for *ccal*, *macro*, *pasm*, and *pcomp*. The intrinsic characteristic of stack buffers to provide a variable amount of storage (up to the full buffer) causes this anomaly. A procedure can claim more buffer space than effectively needed. For example, claiming a large part of the buffer, flushing part to memory, and subsequently returning without using the buffer sufficiently causes traffic ratios to occur higher than average (sometimes much higher than one). The anomaly is intrinsic, however, to the variable-sized procedure windows in *stack* buffers for **memory-to-memory** architectures. Even *stack(single,global)* exhibits this behavior for *pasm*, and for small buffers ($\leq$ 32 registers) in *macro* and *pcomp*.

There are two basic methods to reduce the anomalous effect and to increase performance in general. The first method concerns a more efficient use of the buffer space by minimizing the space required for all procedures. Minimization can be done effectively by means of a *uopt*-like allocator[6] (see Section III-A). In this paper we will not explore this possibility to avoid a bias in the comparison of Section VII, because both *mrs(inter)* and *srs(inter)* also do not exploit such a global allocation strategy. The second method concerns reduction of the under- and overflow traffic.

[5] Interprocedural allocation has limited effects when the set size is small.

[6] The allocator must exploit a strategy which minimizes the required space by assigning data live-ranges to registers instead of variable-ranges (the life-time of a procedure).

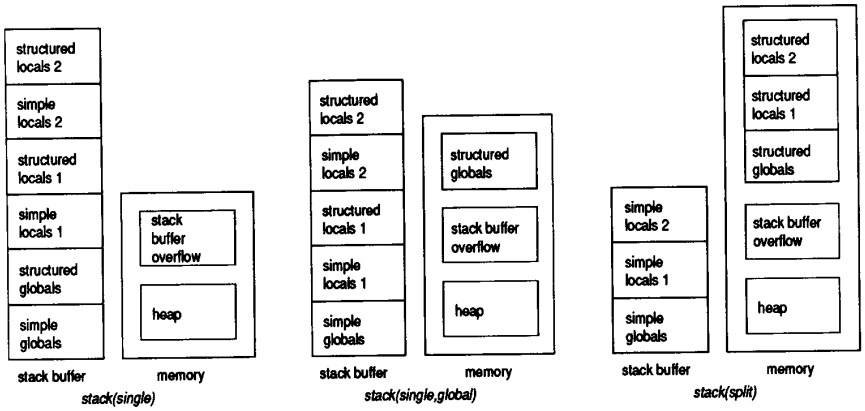


Fig. 9. Three **memory-to-memory** architectures with buffered run-time stacks.

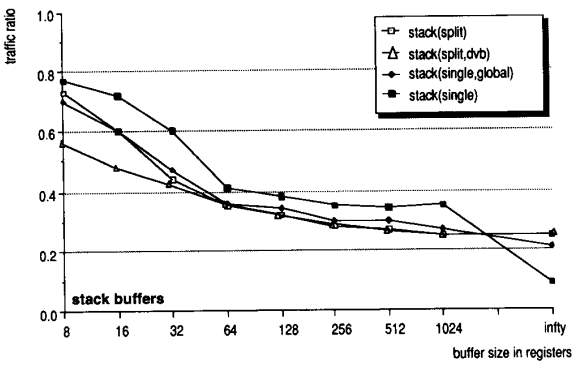


Fig. 10. Traffic ratio for *stack* organizations.

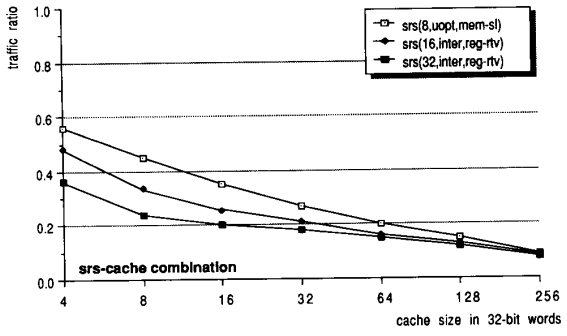


Fig. 11. Traffic ratio as function of cache size and number of registers.

In contrast with *mrs* and *srs* organizations, an interprocedural allocation strategy has no advantage for *stack* because the buffer space utilization on a procedural basis is already optimal and comparable with *srs* and *mrs*. Instead of by software the most obvious way to reduce under- and overflow traffic is to keep track of both validity and modification status of every buffer element. Attaching a valid bit to every buffer entry avoids unnecessary loads because variables are loaded when requested instead of being prefetched upon underflow. Attaching a dirty bit, reflecting modification of the buffer entry, avoids unnecessary stores because upon overflow only the entries modified require a store. As can be expected such a scheme only affects performance for smaller buffers (8–16 words) and significantly reduces the possibility of anomalous behavior for all buffers. Subsequent buffer comparisons assume such a mechanism (or software causing a similar effect); the notation for a stack buffer including this mechanism is *stack(split,dvb)*. See [5] for a more detailed discussion on the use of dirty and valid bits.

VI. Register-Set and Cache Combination

Instead of implementing scalability in the first-level of the buffer hierarchy it might be more efficient to fix the first-level and implement scalability in the second level by means of a cache [17]. Introducing a cache as part of the memory hierarchy instead of a scaling the first-level buffer complicates the comparison of buffering alternatives. The chip area required for a cache and a register file of the same size are different. In this paper we will not take this difference into account [18]. A cache does a good job reducing traffic, but, unfortunately, reduced traffic does not necessarily imply a better performance. In this paper we will take this into account by using the *cycle ratio* in addition to *traffic ratio* (see Section I-A).

The cache used in this combination is a fully associative cache with a line size of two 32-bit words. The cache replaces lines according to their recent use (least recently used) and employs a writeback scheme, meaning, that lines modified during their existence in the cache are only written back upon replacement of the line. To reduce the traffic overhead caused by fetching two words (the whole line) when only a single word (the request) is required, the cache employs a subblock-placement strategy implying a single read for every miss, but possibly incurring two misses to obtain the contents of a whole line [19], [20]. Subblock placement causes subblock allocation on a word-write miss, but no fetch.

Fig. 11 shows the traffic of the register-cache combination for three different set-allocator pairs. The effect of the architecture (number of registers and register-allocator) on data-cache performance is similar to the effect of the architecture on instruction-cache performance as described by

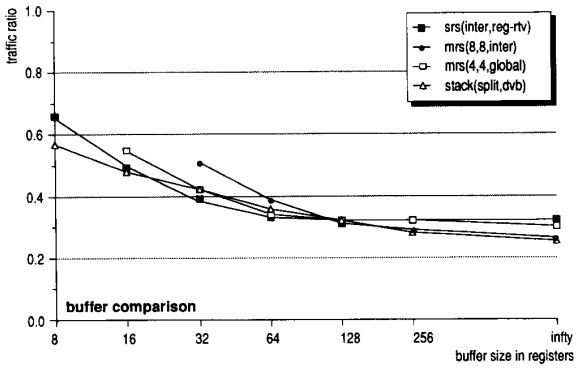


Fig. 12. Data traffic relative to **stack** as a function of architecture-buffer combinations.

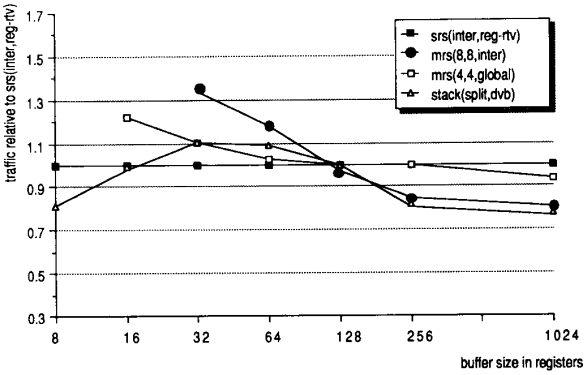


Fig. 13. Data traffic of *mrs* and *stack* relative to *srs(inter)* with identical buffer sizes.

Mitchell [21]. Small caches pronounce the difference between the architectures, while for large caches the differences become small. The traffic ratio for all three register-allocator pairs is approximately 10% for a 256-word cache. This traffic ratio is significantly better than the ratios of the buffers presented in the previous three sections. As we will show in the following section, however, comparing register-oriented buffers with cache buffers requires a different measure than traffic ratio.

VII. Architecture Comparison

This section compares the data of the different architectures. The comparisons relate to four specific questions:

1) How do all architectures compare to the reference architecture (without any data buffering)?
2) Is compile-time software able to achieve the same data traffic ratios as run-time hardware?
3) What are the minimal requirements with respect to buffer size for scalable memory organizations? And what is the performance price of unscalable memories when technology allows implementation of larger buffers?
4) What is the answer to the last question when the unscalable memories are combined with a small data cache?

Finally we assess the applicability of these results to application with a strong vector-reference component.

A. Traffic Relative to **stack**

Fig. 12 summarizes the average traffic ratio for four architectures with different buffer sizes. The table shows the significant advantage of *srs(inter)* for small memories ($\leq$ 16 registers). Buffers of 32 registers show close-to-equal performance for *srs(inter)*, *stack(split,dvb)* and *mrs(4,4,yes)*. Despite the interprocedural allocation *mrs(8,8)* cannot quite keep up with the other buffers for small sizes ($\leq$ 32 registers). The anomalous behavior of *stack(split)* is masked by the dirty-valid bit organization (*dvb*). This behavior only shows when comparing *stack(split,dvb)* and *mrs(4,4)* for 32 and 64 registers where *stack* should outperform *mrs* but does not. Because the over- and underflow overhead of *mrs(8,8,yes)* is still high for 64 register buffers, it takes 128 registers to make all large-buffer organizations perform equally well. Buffers larger than 128 registers finally reach their expected[7] ordering: first *stack(split,dvb)*, followed by *mrs(8,8,yes)*, *mrs(4,4,yes)* and *srs(inter)*. Of the 25% to 30% of the traffic that remains most is due to heap and structure traffic.

B. Compile-time Software Versus Run-time Hardware

Fig. 13 shows the comparison between data traffic reduction by compile-time software only (*srs(inter)*), a mix of compile-time software (register allocation) and run-time hardware (*mrs*), and run-time hardware only (*stack* buffer) organizations. Under the workload used for this study, a single register set with interprocedural register allocation (*srs(inter)*) outperforms both *mrs* organizations for buffers until a size 64 registers, and *stack(split,dvb)* between 16 and 64 registers. As shown before, all buffer organizations containing 128 registers perform equally.

The ability to determine the potential usage of a data item causes the advantage of *srs(inter)* for small buffers. With small buffers the *mrs* and buffered *stack* architectures suffer from over- and underflow penalties because of their inefficient register use; *mrs* because registers are saved even if they are unused, and buffered *stack* because space is allocated for infrequently used variables.

When the over- and underflow penalty is reduced, the intrinsic advantages of *mrs* and buffered *stack* appear. These architectures capture references in recursive procedures and through pointers, while *srs* organizations force those to memory. Because the performance of buffered **memory-to-memory** architectures is not limited by window or set size, larger buffers show the advantage of *stack(split)* over *mrs(8,8,yes)*. Larger buffers also show the advantage of *mrs(8,8,yes)* over *mrs(4,4,yes)*.

C. Scalable Versus Fixed Local Memory

Fig. 12 shows the limitation of *mrs* and *stack* architectures to perform when only limited buffer space is available. Both *mrs(4,4,yes)* and *stack(split)* require at least 32 registers, and

[7] Based on the possible reference coverage as presented in the previous sections.

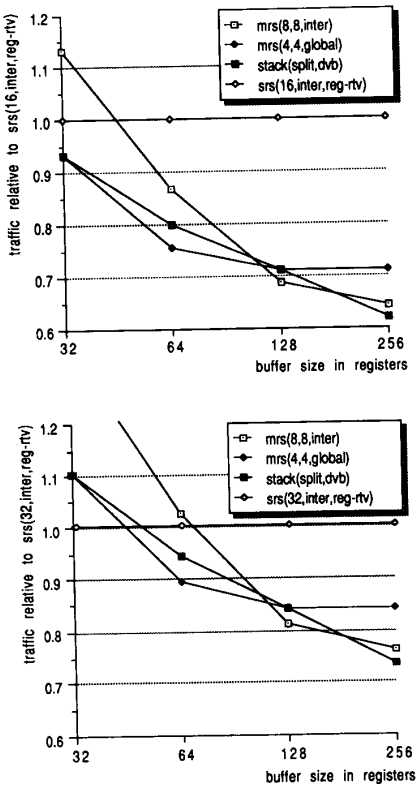


Fig. 14. Data traffic of *mrs* and *stack* architectures relative to *srs(16,inter,reg-rtv)* and *srs(32,inter,reg-rtv)*).

mrs(8,8,yes) requires at least 128 registers. Scalable architectures are probably not the architecture of choice when a range of implementation technologies is required which include low density ones (e.g., GaAs, ECL).

Fig. 14 shows the comparison between two unscalable architectures (*srs(16,inter,reg-rtv)* and *srs(32,inter,reg-rtv)*) and larger scalable architectures (*mrs* and *stack*). As expectable both unscalable organizations perform closely for 32-register buffers. With respect to *srs(16,inter)*, the scalable organizations have an advantage of approximately 20% for 64 registers 30% for 128 registers and 35% for 256 registers. With respect to *srs(32,inter)*, these figures are approximately 10% for 64, 20% for 128, and 25% for 256 registers.

Note that a 20% to 30% traffic reduction for 224 registers is an expensive improvement, especially when taking into account the potential increase in cycle time and the much smaller effect on the number of execution cycles. When memory access time is long relative to the cycle time of a processor, traffic ratio is a reasonable performance indicator. Many current processors, however, rely on single-cycle cache accesses, implying a less pronounced performance effect of the traffic differences presented above. For example when after sufficient buffering only one in five instructions accesses memory, a first order approximation translates a 20% traffic change in only a 5% instruction change. A 5% instruction change, again, yields a smaller performance change because

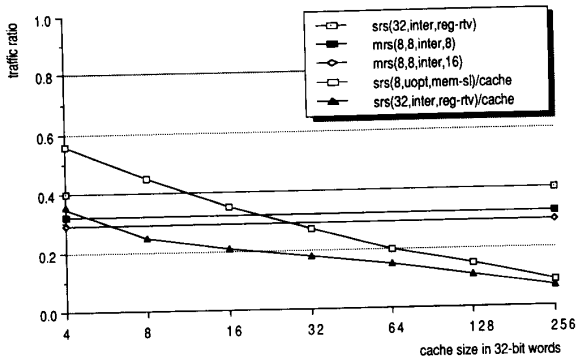


Fig. 15. Traffic comparison of *srs-cache* with *mrs* and *srs*.

the total number of cycles executed is higher than the total number of instructions.

D. Scalable Local Memory Versus a srs-Cache Combination

When comparing traffic only it is not surprising to find the srs-cache combination to outperform *mrs* organizations. 256 words of cache in addition to *srs(32,inter,reg-rtv)* results in a traffic ratio close to 10% (see Fig. 15).

Traffic is not the right measure to compare two level memory hierarchies (buffer, main memory) with three level hierarchies (buffer, cache, memory). The following figures will demonstrate this by showing both traffic and cycle ratio as defined in Section I-A The cycle ratio models the number of cycles spent in the memory hierarchy relative to these cycles for the **stack** architecture.

When calculating the cycle ratio we have to take the memory speed into account. Fig. 16 shows this ratio for a 2-cycle and a 4-cycle memory access. Assume very large memory access times will make the cycle ratio figures approach those of the traffic ratio. A 2-cycle memory closes the gap between *mrs* and *srs-cache* organizations. These memories also introduce a gap between *srs(8,uopt,mem-sl)* and the other organizations because the inferior first-level performance. It is interesting to see that it is very difficult to compensate first-level deficiencies by means of a cache. A cache in combination with *srs(32,inter,reg-rtv)*, however has sufficient first-level performance to outperform all other buffers with only a small cache (> 16 words).

Slower memories (Fig. 16, 4-cycle memory) advance both cache combinations because the second-level buffer (cache) profits particularly from slow access times. Now *srs(inter)/cache* outperforms the register-only hierarchies for caches of 8-words and larger. *Srs(uopt)/cache* requires 64-words of cache to compete with the register-only hierarchies.

E. Applicability of Results to Other Application Areas

The programs used for this study are representative of a typical general-purpose workstation workload. All applications have a symbolic-processing character, the dominant datatype is scalar integer, and the data locality depends mainly on the problem and not on the size of the input set. The programming

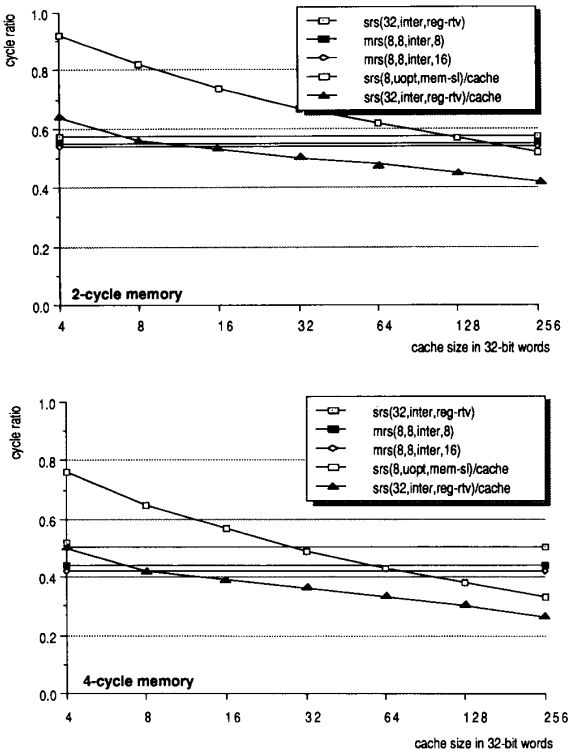


Fig. 16. Cycle-ratio comparison of *srs-cache* with *mrs* and *srs*.

language used is largely irrelevant; the results presented also hold, for symbolic applications written in Lisp and Prolog [5].

Scientific and engineering programs, however, do have different characteristics. Compared with the general-purpose programs used in this paper differences are: 1) shallower call chains, 2) no recursion, 3) data references dominated by vector instead of scalar reference patterns. The first two issues indicate that buffering scalar data is potentially easier because the influence of procedure calls on performance is smaller. Most likely, interprocedural-allocation and *mrs* or *stack* schemes provide functionality not really required. The third item implies the reduced importance of the CPU and its buffering strategy. Buffering structured (vector) data requires large vector registers and/or pipelined caches and memories. Overall the locality will be less than found in the general-purpose programs, the locality of scalar references however will be higher, and the locality of vector references will depend strongly on the size of the input data.

Fig. 17 shows a comparison between a workstation (general purpose) and a scientific[8] workload for both *srs* and *mrs*. The figure clearly shows the assumptions correct: *srs(16,uopt,mem-sl)* captures close to all scalar references for the scientific load, but runs into the call-return barrier for the workstation load; *mrs(8,8,4(64))* crosses this barrier for the workstation load, but adds very little for the scientific load. A smarter compiler, most likeley, will reduce the amount of scalar traffic

[8] 256-point complex fft, kalman, and a (twice unrolled) loop11 of the livermore loops.

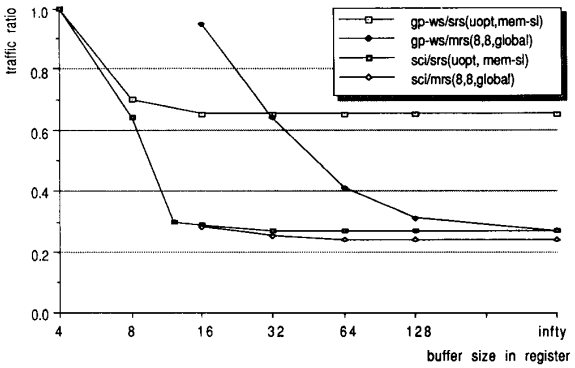


Fig. 17. Comparison between scalar and scientific workloads.

and emphasize the conclusion that a simple *srs* offers the best cost/performance for a scientific workload. The issue of caching the remaining vector references is a research topic on itself. A small cache capturing both scalar and vector references, will only harm the performance because of the different reference patterns and strongly reduced locality of vector code. A small cache just covering scalars is unlikely to contribute significantly to the performance.

VIII. Conclusions

The design of processor architectures is a continuous tradeoff between functionalities and their potential performance effects. The importance of the right architecture tradeoffs increases with increasing restrictions imposed by the technology of choice. VLSI technologies impose implementation restrictions on, for example, the area, the I/O bandwidth and the power consumption. This paper discusses tradeoffs in processor architecture, but restricts the scope of these tradeoffs to the effect of architecture and local memory on data-memory traffic.

We have investigated data-traffic characteristics of three architecture families: **stack**, **register**, and **memory-to-memory** architectures. The architecture parameters under consideration solely concern local memories and, when applicable, the compile-time software exploiting these memories. The local-memory organizations investigated are an evaluation stack for **Stack**, a single register set (*srs*) and multiple overlapping register sets (*mrs*) for **register**, and a stack buffer or stack cache for **memory-to-memory**. An additional data point concerns a combination of a small register set with a small cache (*srs-cache*). The comparison of the different architectures yields three main conclusions.

First, with respect to software-hardware tradeoffs: Compile-time software (register allocation) outperforms run-time hardware organizations quite easily for small buffers ($\leq$ 32 registers), performs approximately equal for buffers up to 128 registers, and performs only slightly below ($\leq$ 10%) these organizations for larger buffers. In addition, some *mrs* organizations require compile-time assistance to compete with *srs* organizations, and stack buffered *stack* organizations require compile time assistance or additional hardware complexity

(dirty and valid bits) to avoid anomalous behavior and to perform competitively.

Second, with respect to scalability tradeoffs: Unscalable architectures (*srs*) incur only a slight disadvantage when compared to scalable architectures with significantly larger buffers. Compile-time exploitation of 32 registers yields an approximate traffic disadvantage of 10% over scalable organizations with 64 registers and only 25% over scalable organizations with 8 times the registers (256). For a 16-register architecture these figures are 30% and 40%, respectively. When translated into cycles, it is unlikely that the cycle advantage of buffer scalability is worth the increase in chip/board area, the potential increase in buffer access time, or the potential increase in processor-cycle time.

Third, when instead of a large number of registers a small register set is combined with a cache then the resulting combination performs significantly better (in cycles) than the large register-oriented buffers. Even for relatively fast memories (2 cycle access time), it is preferable to exploit a 32-register set allocated interprocedurally and combined with a small cache (e.g. 64 words). Larger caches and slower memory access times increase the advantages of *srs-cache* organizations.

When substituting vector programs for the scalar programs used, the results point even stronger to the use of *srs* instead of more complex *mrs* or *stack* organizations.

REFERENCES

[1] D. R. Ditzel and H. R. McLellan, "Register allocation for free: The C Machine stack cache," in *Proc., Symp. Architectural Support for Programming Languages and Oper. Syst.*, Mar. 1982, pp. 48–56.

[2] A. D. Berenbaum, D. R. Ditzel, and H. R. Mclellan," "Introduction to the CRISP instruction set architecture," in *Proc. COMPCON 1987*, Jan. 1987, pp. 86–90.

[3] D. Alpert, "Memory hierarchies for directly executed language microprocessors," Tech. Rep. 84-260, Comput. Syst. Lab., Stanford Univ., Stanford, CA 94305, June 1984.

[4] D. A. Patterson and C.H. Sequin, "RISC I: A reduced instruction set VLSI computer," in *Proc. Eight Int. Symp. Comput. Architecture*, IEEE and ACM, May 1981.

[5] J. M. Mulder, "Tradeoffs in processor-architecture and data-buffer design," Tech. Rep. 87-345, Comput. Syst. Lab., Stanford Univ., Stanford, CA 94305, Dec. 1987.

[6] C. L. Mitchell and M. J. Flynn, "A workbench for computer architects," *IEEE Design & Test*, vol. 5, no. 1, pp. 19–29, Feb. 1988.

[7] B. Bray, K. Cuderman, M. Flynn, and A. Zimmerman, "The computer Architect's Workbench," in *Information Processing 89 (IFIP)*, G. X. Ritter, Ed. Amsterdam, The Netherlands: Elsevier Science Publishers B. V., (North-Holland), Sept. 1989.

[8] F. C. Chow, "A portable machine-independent global optimizer–Design and measurements," Tech. Rep. 83-254, Comput. Syst. Lab., Stanford Univ., Stanford, CA 94305, Dec. 1983.

[9] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, . E. Hopkins, and P. W. Markstein, "Register allocation via coloring," *Comput. Languages*, vol. 6, pp. 47–57, 1981.

[10] F. Chow and J. Hennessy, "Register allocation by priority-based coloring," in *Proc. SIGPLAN'86 Symp. Compiler Construction*, Montreal, Canada, ACM, June 1984, pp. 222–232.

[11] D. Wall, "Global register allocation at link time," in *Proc. SIGPLAN'86 Symp. Compiler Construction*, ACM, June 1986, pp. 264–275.

[12] P. Steenkiste and J. Hennessy, "A simple interprocedural register allocation algorithm and its effectiveness for lisp," *Trans. Programming Languages and Syst.*, vol. 11, no. 1, pp. 1–32, Jan. 1989.

[13] F. C. Chow, "Minimizing register-usage penalty at procedure calls," in *Proc. SYSPLAN '88: Conf. Programming Language Design and Implementation*, June 1988, pp. 85–94.

[14] M. G. H. Katevenis, " Reduced instruction set computer architectures for VLSI," Ph.D. dissertation, Comput. Sci. Division (EECS), Univ. of California Berkeley, Oct. 1983.

[15] D. C. Halbert and P. B. Kessler, "Windows of overlapping register frames," Tech. rep., Comput. Sci. Division, Univ. of California Berkeley, CA 94720, 1980.

[16] J. M. Mulder, "Data buffering: Software versus hardware support," in *Proc., Conf. Architecture Support for Programming Languages and Oper. Syst.*, Apr. 1989, pp. 144–151.

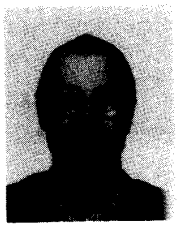
[17] M. J. Flynn, C. L. Mitchel, and J. M. Mulder, "And now a case for more complex instructions," *IEEE Comput. Mag.*, vol. 20, no. 9, pp. 71–83, Sept. 1987.

[18] J. M. Mulder, N. T. Quach, and M. J. Flynn, "An area model for on-chip memories," *J. Solid State Circuits*, vol. 26, no. 2, pp. 98–106, Feb. 1991.

[19] M. D. Hill and A. J. Smith, "Experimental evaluation of on-chip microprocessor cache memories," in *Proc. 11th Annu. Int. Symp. Comput. Architecture*, June 1984, pp. 158–166.

[20] M. J. Flynn and D. Alpert, "Performance trade-offs for microprocessor cache memories," *IEEE Micro*, vol. 8, no. 4, pp. 44–54, Aug. 1988.

[21] C. Mitchell and M. Flynn, "The effects of processor architecture on memory traffic," *Trans. Comput. Syst.*, vol. 8, no. 3, pp. 230–250, Aug. 1990.



Hans Mulder (S'82–M'87) received the M.S. degree from Delft University of Technology and the Ph.D. degree from Stanford University.

He is an Assistant Professor in the Electrical Engineering Department of Delft University of Technology. His main research interests are computer architecture, compiler, and VLSI design for high-speed computing, and computer-aided architecture and system design. He is the principal investigator of the SCARCE project, which concerns the design of application-specific processors for high-speed embedded controllers.

Dr. Mulder is a member of the IEEE Computer Society and the Association for Computing Machinery.

Michael J. Flynn (M'56–SM'79–F'80) was born in New York City on May 20, 1934. He received the B.S.E.E. degree from Manhattan College in 1955, the M.S. degree from Syracuse University in 1960, and the Ph.D. degree from Purdue University in 1961.

He joined IBM in 1955 and, for ten years, worked in the areas of computer organization and design. He was design manager of prototype versions of IBM 7090 and 7094/11, and later was design manager for the System 360 Model 91 Central Processing Unit. He was a faculty member of Northwestern University (1966–1970) and the Johns Hopkins University (1970–1974). In 1973–1974 he went on leave from Johns Hopkins to serve as co-founder and Vice President of Palyn Associates, Inc.—a computer design firm in San Jose, CA, where he is now a senior consultant. In January 1975, he became Professor of Electrical Engineering at Stanford University, and was Director of the Computer Systems Laboratory from 1977 to 1983.

Dr. Flynn has served on the IEEE Computer Society's Board of Governors and as an Associate Editor of the IEEE TRANSACTIONS ON COMPUTERS. He has also served as consultant and advisor to a number of private and government organizations: he has been consultant to the Army Research Office–Durham, a member of the Scientific and Management Advisory Committee (SAMAC) to the Army Computer Systems Command, and is a member of the Universities Space Research Association (USRA) Science Council at ICASE, Langely.