Stack Cache Memory for Block-Structured Programs

LANFRANCO LOPRIORE

Dipartimento di Ingegneria della Informazione: Elettronica, Informatica, Telecomunicazioni, Università degli Studi di Pisa, via Diotisalvi 2, 56126 Pisa, Italy

The architecture of a cache memory is presented, aimed at reducing the memory bandwidth requirements of programs written in block-structured, high-level languages. At any given time, the cache contains two portions of the stack area of the running program, corresponding to the global and the local activation records. With respect to traditional cache architectures, the proposed architecture is characterized by increased performance and a reduced complexity of the logic for cache space addressing and management. These results have been obtained by controlling the cache activity at the software level to take advantage of the stack paradigm.

Received December 1993; revised May 1994

1. INTRODUCTION

The advantages resulting from the utilization of a cache memory to decouple the processor from the primary memory are well known (Hennessy and Patterson, 1990; Stone, 1993). In the implementation of a microprocessor architecture, a cache makes it possible to reduce the gap existing between the speed of the microprocessor and that of low cost memory chips (Matick, 1989). In the traditional approach, the cache effectiveness relies on the locality of reference property that characterizes the program references to the address space (Silberschatz *et al.*, 1991). In this approach, the cache treats information items characterized by very different reference patterns, such as code and data objects, by using an internal model, wired in the cache logic, of the program memory behaviour. Cache storage resources are managed according to this model, to select the cache line to be replaced when a miss occurs and no free line is available, for instance (Smith and Goodman, 1985).

In a different approach, the cache has a degree of knowledge of the semantics of the information items it contains. An example is a special-purpose cache designed to support a specific programming language, so as to take advantage of the memory referencing characteristics of programs written in this language (Lopriore, 1993). A cache can be also specialized to contain portions of the program code (Farrens and Pleszkun, 1989; Hwu and Chang, 1989), to exploit the read-only characteristics of the instructions by not restoring the state of the primary memory when a line is replaced, for instance. In this way, the operations connected with cache space management are simplified. The ensuing savings of cache logic are especially important if the cache is on-chip with the processor, as more chip area can be used for the cache line array to increase the cache capacity, or even for other processor functionalities, such as pipelining, floating point, or the register file. In fact, instruction caches are now of widespread use in microprocessor implementations (Kane and Heinrich, 1992; Slater, 1992).

In this approach, we have investigated the possibility of reducing the memory bandwidth requirements of programs written in block-structured, high-level languages by means of a data cache specialized to contain information items belonging to the stack area of the program address space. In the implementation of a block-structured language, the program space is usually partitioned into three areas, the *code*, the *heap* and the *stack* (Ghezzi and Jazayeri, 1987; Pratt, 1984). The code is the static program area consisting of the machine instructions generated from the statements in the program, and those program components that are invariant during execution, e.g. the representations of the literals and the program-defined constants. The heap is the dynamic program area aimed at containing those data objects whose size can vary at arbitrary program points, e.g. flexible arrays and dynamic variables that can be allocated and destroyed under program control. Finally, the stack is the dynamic area containing the program *activation records* (also called *stack frames*; Watt, 1993). An activation record contains the data items needed for an activation of a subprogram at run time. Examples are the subprogram parameters, the local variables, and the temporaries in expression evaluation and in the transmission of parameters to other subprograms.

At the beginning of the program execution, the stack contains a single activation record, the *global* activation record relevant to the environment of the main program block. This activation record includes space for the global variables. It is stored at the stack base, and is invariant for the entire duration of the program execution. When a subprogram is entered, a new activation record is allocated onto the stack for this subprogram. The activation record will be deleted on subprogram termination. The *local* activation record is the activation record of the subprogram activation being executed, stored at the stack top and containing the local

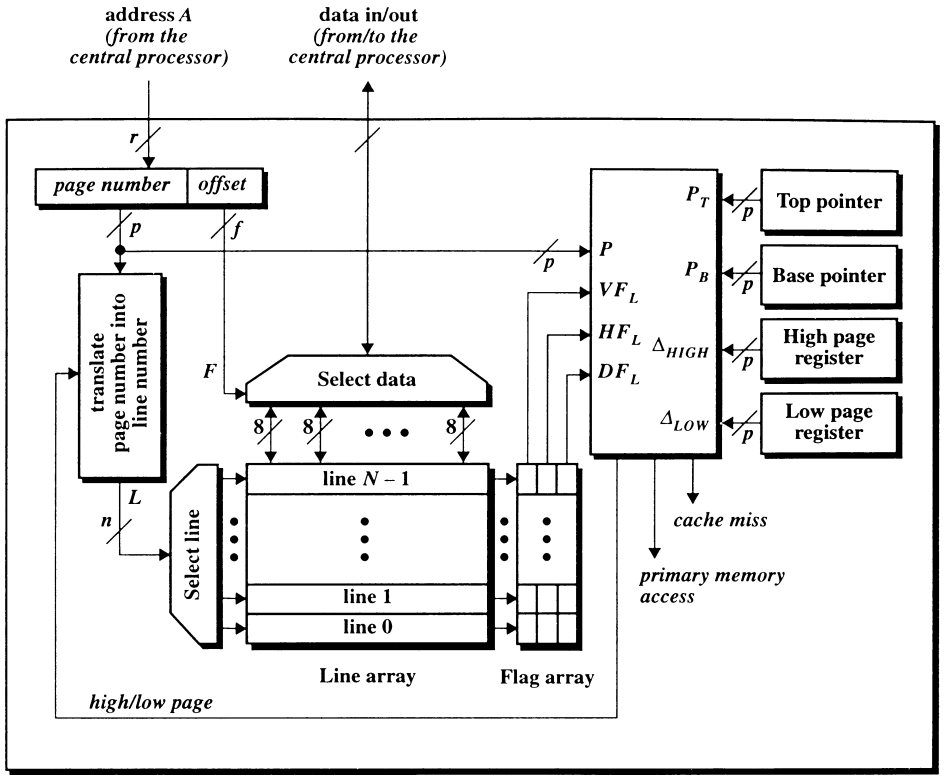

FIGURE 1. Block diagram of the cache hardware.

variables. The *non-local* activation records are the activation records relevant to the other activations, which are stored at intermediate stack positions between the local and the global activation records, and contain the non-local variables.

Experimental studies indicate that the accesses to the stack area are characterized by a high locality of reference in the local and the global activation records (Carter, 1982; Cook and Lee, 1982). Thus, a data cache which can store both these activation records is likely to capture a large fraction of the stack references, and the miss ratio of this cache will be low. Measurements show that the memory requirements of these activation records are moderate (Batson and Brundage, 1977; Ditzel, 1980). It is therefore viable to store both of them in the cache even in an on-chip implementation, provided that less chip space is used to implement the functionalities of cache storage management and addressing, and more space is reserved for the cache line array to increase the cache capacity.

In the following sections, we will present the architecture of a stack cache able to take advantage of the stack paradigm to enhance performance and reduce the hardware costs. In contrast to classical cache architectures, our architecture takes advantage of a software control over the cache operations, which is exercised by means of a set of cache commands. These commands will be inserted by the compiler at appropriate points in the program object code to transmit the cache information available at translation time, concerning the program memory behaviour.

2. ARCHITECTURE OF THE CACHE

We will refer to a system architecture featuring a cache memory interposed between the central processor and the primary memory. The processor references an address space of size 2^r bytes. This address space is partitioned into 2^p *pages* and the size of a page is 2^f bytes, $f = r - p$. The cache storage is organized into an array of $N = 2^n$ lines and the line size is equal to the page size. At any given time, the cache contains up to N pages of the data space of the running program. These pages are always relevant to the program stack area.

Figure 1 shows the block configuration of the cache hardware. When the cache receives an r-bit memory address A from the central processor, the *page number* P and the *offset* F corresponding to that address are evaluated by selecting the p most significant bits and the f least significant bits of the address, respectively. The page number is translated into the number L of the corresponding line in the cache line array and then the offset is used to identify the referenced data item within this line.

2.1. Translating page numbers into line numbers

In the following, we will hypothesize that the stack grows upwards, towards increasing memory addresses, and that activation records are stored at page boundaries. Figure 2 shows the program stack area. The *local pages* are the pages of the local activation record, the *global pages* are the pages of the global activation record, the *top page* is the local page containing the highest stack

address (the stack *top*) and the *base page* is the global page containing the lowest stack address (the *base*). We will denote the stack size, in pages, by Δ_{STK}, and the sizes of the local and the global activation records by Δ_{LAR} and Δ_{GAR}, respectively. Let P_T and P_B be the page numbers of the top and the base pages. Then

$$\Delta_{STK} = P_T - P_B + 1 \quad (1)$$

Translation of page numbers into line numbers is obtained by using a classification of the stack pages into *high* and *low* pages. Let

$$\Delta_{HIGH} = \min(N, \Delta_{LAR}) \quad (2)$$

denote the minimum between quantities N and Δ_{LAR}, and let

$$\Delta_{LOW} = \min(N, \Delta_{STK} - \Delta_{LAR}) \quad (3)$$

denote the minimum between quantities N and $\Delta_{STK} - \Delta_{LAR}$. The high pages are the local pages at a distance from the top page shorter than Δ_{HIGH} pages, i.e. each page whose number P_{HIGH} satisfies relation

$$P_T - \Delta_{HIGH} + 1 \leqslant P_{HIGH} \leqslant P_T \quad (4)$$

The low pages are the stack pages which are not high pages and are at a distance from the base page shorter than Δ_{LOW} pages, i.e. each page whose number P_{LOW} satisfies relation

$$P_B \leqslant P_{LOW} \leqslant P_B + \Delta_{LOW} - 1 \quad (5)$$

It should be noted that a local page is never classified as low, even if it is at a short distance from the base.

At any given time, only the high and the low pages are candidate to be stored in the cache. The cache lines are managed circularly, line $N-1$ being considered adjacent to line 0. Let $|P|_N$ denote the result of a modulus operation between quantity P and the cache capacity N, and let L_B denote quantity $|P_B|_N$. The line number L corresponding to a given page number P is given by relation

$$L = \begin{cases} |P|_N & \text{if } P_B \leqslant P \leqslant P_B + \Delta_{LOW} - 1 \quad \text{(i.e. } P \text{ is a low page)} \\ |L_B - (P_T - P) - 1|_N & \text{if } P_T - \Delta_{HIGH} + 1 \leqslant P \leqslant P_T \quad \text{(i.e. } P \text{ is a high page)} \end{cases} \quad (6)$$

It follows that the base page P_B is stored in line L_B, the low pages are stored from line L_B towards increasing line numbers, the top page P_T is stored in line $L_T = |L_B - 1|_N$ and, finally, the high pages are stored from line L_T towards decreasing line numbers.

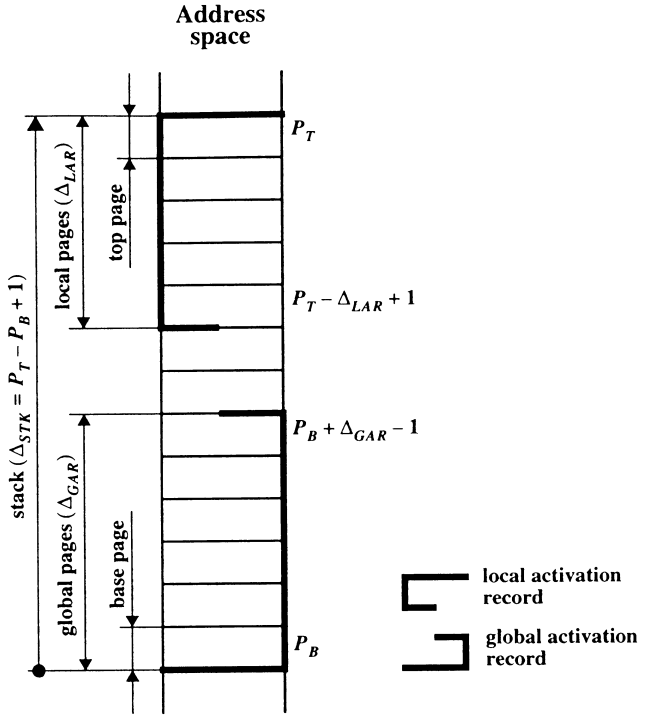

FIGURE 2. Address space configuration for the stack area.

2.2. Configurations

With reference to a cache size N of 8 pages, Figure 3 shows the high and the low pages in a number of different configurations for the stack area. The configurations of Figure 3(a–d) are all relevant to situations in which $\Delta_{STK} - \Delta_{LAR} \geqslant N$; in these situations, $\Delta_{LOW} = N$ and $P_B + \Delta_{LOW} - 1 = P_B + N - 1$. In the configuration of Figure 3(a), both the local and the global activation records are smaller than N pages. It follows that $\Delta_{HIGH} = \Delta_{LAR}$, all the local pages are high pages, all the global pages are low pages and there are non-local pages near the stack base that are classified as low. In the configuration of Figure 3(b), the local activation record is larger than N pages. In this case, $\Delta_{HIGH} = N$ and only the N local pages at high stack addresses are high pages candidate to be stored in the cache (the other local pages will never be loaded). In Figure 3(c), the global activation record is larger than N pages. It follows that only the N global pages at low stack addresses are low pages and can be stored in the cache. In Figure 3(d), both the global and the local activation records are larger than N pages. In a situation of this type, the local pages at low stack addresses and the global pages at high stack addresses will never be stored in the cache. The configurations of Figure 3(e and f) are both relevant to situations in which $\Delta_{STK} - \Delta_{LAR} < N$; in these situations, $\Delta_{LOW} = \Delta_{STK} - \Delta_{LAR}$ and $P_B + \Delta_{LOW} - 1 = P_T - \Delta_{LAR}$ (see relation 3). In Figure 3(e) both the local and the global activation records are smaller than N pages; whereas in Figure 3(f) the local activation record is larger than N pages. In both cases, the local pages are

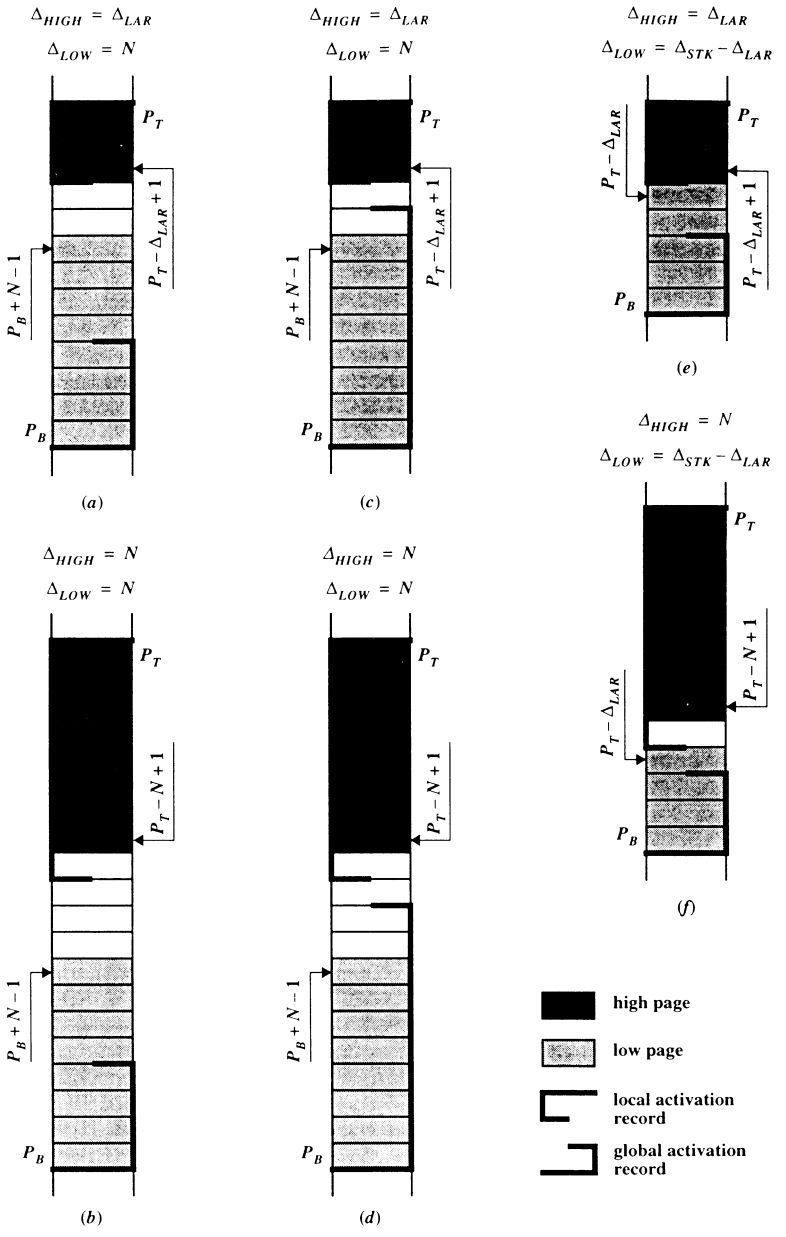

FIGURE 3. The high and the low pages in a number of stack configurations.

classified as high even if they are at a distance from the stack base shorter than N pages.

Figure 4–6 show the cache line array in a number of significant address space configurations. In the configuration of Figure 4, $\Delta_{STK} \leqslant N$, and the entire stack can be contained in the cache. In a situation of this type, $\Delta_{HIGH} = \Delta_{LAR}$ and $\Delta_{LOW} = \Delta_{STK} - \Delta_{LAR}$ (see relations 2 and 3). All the local pages are high pages; these pages are stored in lines $|L_B - \Delta_{HIGH}|_N = |L_B - \Delta_{LAR}|_N$ to L_T. All the other stack pages (including the global pages) are low pages; these pages are stored in lines L_B to $|L_B + \Delta_{LOW} - 1|_N = |L_B + (\Delta_{STK} - \Delta_{LAR}) - 1|_N$. Lines $|L_B + (\Delta_{STK} - \Delta_{LAR})|_N$ to $|L_B - \Delta_{LAR} - 1|_N$ are free. In the configuration of Figure 5, $N < \Delta_{STK} \leqslant N + \Delta_{HIGH}$, the stack can be contained in the cache only partially and all the stack pages are high or low pages candidate to be stored in the cache. The local pages from $P_T - \Delta_{HIGH} + 1$ to $P_B + N - 1$ are at a distance from the stack base shorter than N lines; as seen previously, these pages are classified as high pages. It follows that lines $|L_B - \Delta_{HIGH}|_N$ to $|L_B + \Delta_{LOW} - 1|_N$ are shared between the high and the low pages. Finally, in Figure 6, , $\Delta_{STK} > N + \Delta_{HIGH}$, the stack can be contained in the cache only partially, and the stack pages $P_B + \Delta_{LOW}$ to $P_T - \Delta_{HIGH}$ cannot be stored in the cache. Lines $|L_B - \Delta_{HIGH}|_N$ to $|L_B - 1|_N$ are shared between the high and the low pages.

3. ACCESSING THE CACHE

3.1. Registers and flags

Two cache registers, the *top pointer* and the *base pointer*, are aimed at containing the numbers P_T and P_B of the

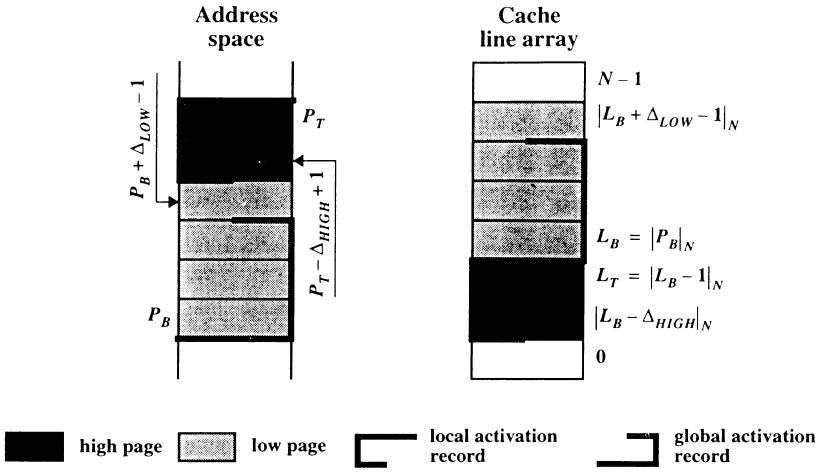

FIGURE 4. Configuration of the address space and the cache line array when the stack size is smaller than or equal to the cache size.

top and the base stack pages (see Figure 1). Two further registers, the *high page register* and the *low page register*, contain quantities Δ_{HIGH} and Δ_{LOW}, respectively.

Three flags, the *validity flag* VF_L, the *high flag* HF_L and the *dirty flag* DF_L, are associated with each given cache line L (Figure 7). If asserted, the validity flag VF_L specifies that L contains valid information. In a valid, shared cache line, the high flag HF_L specifies whether this line contains a high page (HF_L asserted) or a low page (HF_L clear). Finally, the dirty flag DF_L determines the memory situation of the page P corresponding to line L in relation to the value of VF_L. As shown in Table 1, if VF_L is asserted and DF_L is clear, both L and the primary memory contain valid information for page P; in a situation of this type, the contents of L match that of the primary memory. If both VF_L and DF_L are asserted, line L contains valid information for page P that are not stored in the primary memory (i.e. the contents of L have been modified since the load of P into the cache). If VF_L is clear and DF_L is asserted, the primary memory contains valid information for P that are not stored in the cache. Finally, if both VF_L and DF_L are clear, page P contains no valid information.

When a valid line L is invalidated, if its dirty flag DF_L is asserted, the line contents must be copied to the primary memory, to avoid the losing of the valid information stored in that line; however, this copy action is not required if the dirty flag is clear. In this way, the dirty flags allow us to save primary memory accesses on line invalidation. In contrast to classical cache architectures, our architecture associates a significant value with the dirty flag of a given line even if this line is not valid. As will be clarified later, this feature allows us to save the memory accesses required to load a page storing no useful information the first time this page is referenced.

3.2. Accessing a data item in the cache

When the cache receives an address A from the central

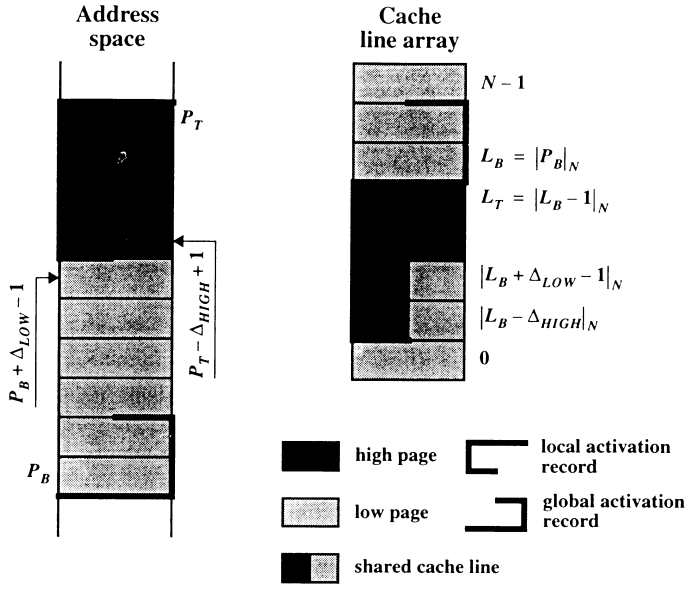

FIGURE 5. Configuration of the address space and the cache line array when the stack is larger than the cache and every stack page can be stored in the cache.

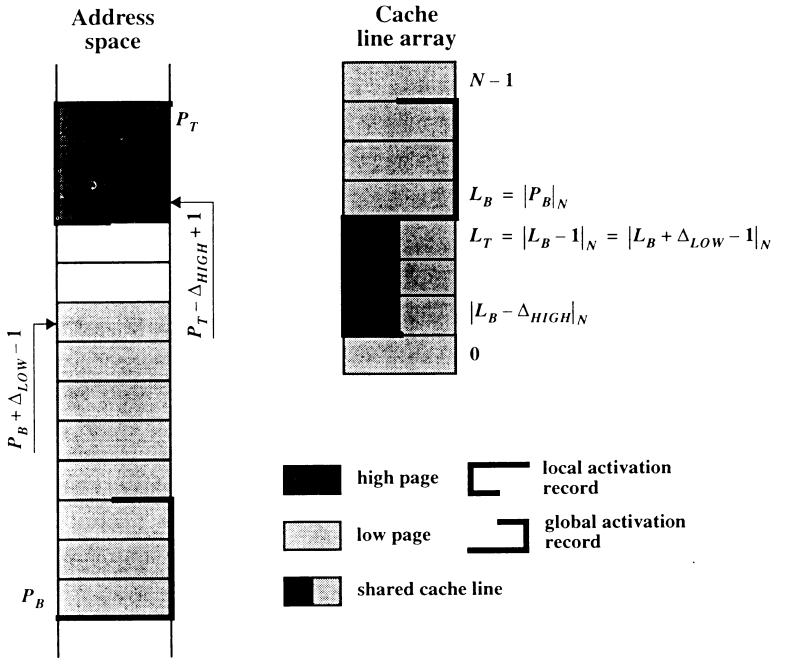

FIGURE 6. Configuration of the address space and the cache line array when the stack is larger than the cache and there are stack pages that cannot be stored in the cache.

processor, the first action is to ascertain whether the corresponding page is a high or a low page and, consequently, can be stored in the cache. To this aim, the page number P is compared with the contents P_T and P_B of the top and the base pointers as well as with the contents Δ_{HIGH} and Δ_{LOW} of the high and the low page registers, according to relations (4) and (5). The results of these comparisons determine the subsequent cache operations, as follows:

1. If the referenced page is either a high or a low page:
   - translate the page number P into the number L of the corresponding cache line (see relation 6);
   - determine whether a cache miss is taking place. We have a miss in the following cases: (i) VF_L is clear and DF_L is asserted (i.e. the primary memory contains valid information for P that are not stored in the cache); (ii) P is a high page, VF_L is asserted and HF_L is clear (i.e. line L contains valid information that belong to a low page); and (iii) P is a low page, and both VF_L and HF_L are asserted (i.e. line L contains valid information which belong to a high page). In case of a miss, load L with quantities taken from the primary memory, set VF_L, clear DF_L and set or clear HF_L, according to the classification (high or low) of page P;
   - if the memory access is for read, send the contents of line L to the data selector, which will then use offset F to extract the referenced data item and return it to the central processor;
   - if the memory access is for write, set the dirty flag DF_L and send the data item incoming from the processor to the data selector, which will then use offset F to store that data item in the corresponding portion of line L.
2. If the referenced page is neither a high nor a low page, it cannot be contained in the cache, and the data access will be carried out in the primary memory.

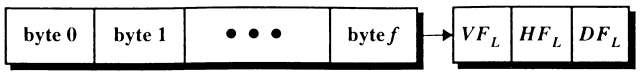

FIGURE 7. A cache line L and its line flags.

4. CACHE COMMANDS

Our cache is able to execute a set of *commands*, which are stored in memory as part of the program code. On fetching a command, the central processor sends it to the cache for execution. In this way, the commands implement a form of direct communication between the running program and the cache. The *Initialize* command clears the cache, the *Expand* and *Shrink* commands update the cache state on the occurrence of a change of the stack size. *Initialize* has the form

Initialize P_B, Δ_{GAR}

This command will be issued at the beginning of the program execution, when the stack contains a single activation record, i.e. the global activation record. Thus, the stack size Δ_{STK} is equal to Δ_{GAR} and $\Delta_{LAR} = 0$. Execution of this command loads the page number of the stack base, as specified by the parameter P_B, into the base pointer, quantity $P_B + \Delta_{GAR} - 1$ into the top pointer, and quantities 0 and $\min(N, \Delta_{GAR})$ into the high and the

TABLE 1.

VF_L	DF_L	*Primary memory*	*Cache (line L)*
clear	clear		
clear	asserted	✓	
asserted	clear	✓	✓
asserted	asserted		✓

The symbol ✓ indicates that the corresponding memory device contains valid information for line L.

low page registers, according to relations (2) and (3). Then, all the cache lines are reserved to contain low pages, by clearing the high flag of each of them. Moreover, as the primary memory contains no valid information for these lines, their validity and dirty flags are cleared.

When execution of a subprogram is started up, a new activation record is inserted onto the stack for this subprogram and the stack expands. The new activation record becomes the local activation record. The *Expand* command allows the running program to notify the cache of the stack growth. This command has the form

Expand Δ_{LAR}

The parameter Δ_{LAR} is the size of the new activation record. Execution considers all the valid lines storing high pages. Each of these lines which is dirty is copied back to the primary memory, and then, all these lines are invalidated, by clearing the validity flag of each of them. The contents of the top pointer are increased by Δ_{LAR}. Then, quantities Δ_{HIGH} and Δ_{LOW} are evaluated according to relations (2) and (3), and are stored into the high and low page registers, respectively. The lines in the range $L_{\mathrm{T}} - \Delta_{\mathrm{HIGH}} + 1$ to L_{T} that do not store low pages are reserved to contain high pages, by setting their high flags. Moreover, as the primary memory contains no valid information for these lines, their validity and dirty flags are cleared.

Finally, when execution of a subprogram terminates, the activation record of this subprogram is extracted from the stack, and the stack shrinks. Control is returned to the previous subprogram, whose activation record now becomes the new local activation record. The *Shrink* command will be used by the terminating subprogram to notify the stack shrinkage to the cache. This command is as follows:

Shrink $\Delta'_{\mathrm{LAR}}, \Delta''_{\mathrm{LAR}}$

The parameters Δ'_{LAR} and Δ''_{LAR} are the sizes of the activation records of the terminating subprogram and of the previous subprogram, respectively. Execution invalidates all the lines storing high pages. The contents of the top pointer are decreased by Δ'_{LAR}. Then, the parameter Δ''_{LAR} is used to evaluate quantities Δ_{HIGH} and Δ_{LOW} and store them into the high and low page registers, respectively. The lines in the range $L_{\mathrm{T}} - \Delta_{\mathrm{HIGH}} + 1$ to L_{T} that do not store low pages are reserved to contain high pages, by setting their high flags. Moreover, as the primary memory contains valid information for these lines, their validity flags are cleared and their dirty flags are asserted.

5. DISCUSSION

5.1. Primary memory update

Classical cache organizations use two different strategies for primary memory update, *store through* (also called *write through*) and *copy back* (Kain, 1989). In the store through approach, when a given cache line is accessed for write, the write is also carried out in the corresponding page of the primary memory. It follows that the primary memory always contains an updated version of the program address space. In the copy-back approach, the write operation is accomplished in the cache and no primary memory access takes place. The line will be copied back to the primary memory only when it is selected for replacement.

Write-through has the advantage that we do not copy an entire line even if only a small portion of this line has been modified, however, a drawback is that the write operations are always as slow as the primary memory. A solution is to provide the cache with a *write buffer* where to store several writes (Smith, 1982). Of course, the copy of the contents of this buffer to the primary memory will use free bus cycles; however, the buffer may increase the time required to service the read misses. Essentially, when a read miss occurs, the page involved in the miss must be fetched from the primary memory only after the buffer has been emptied, to avoid the reading of outdated information if the buffer stores valid data for this page. Alternatively, we may check whether the information contained in the buffer are relevant to the referenced page, or not, and wait for the buffer to be emptied only if the check is successful. This solution has a high hardware cost and is not viable in an on-chip implementation of the cache.

Our cache uses an improved version of copy back aimed at saving write cycles. When execution of a subprogram terminates, the lines reserved for the activation record of this subprogram contain discarded information. As seen in Section 4, the *Shrink* command simply invalidates these lines, and no copy back occurs. This is in contrast to what happens in traditional caches, where a line selected for replacement is always copied to the primary memory, even if it contains no useful information.

5.2. Cache update

When the processor references a data item that is not resident in the cache, the page containing this data item must be loaded from the primary memory into the cache. In a classical copy-back cache, this *fetch-on-miss* must be

performed even if the access is for write, to avoid that, after the access, information for the corresponding page be stored partially in the cache and partially in the primary memory. Of course, every line fetch produces negative effects on both the memory traffic and the processor-to-memory access time, and reduces the performance of the processor, which has to wait for the fetch to terminate.

Our cache takes advantage of the stack paradigm to reduce the number of line fetches. When a new activation record is inserted onto the stack as a consequence of a subprogram entry, the cache contains no page of this activation record. Of course, the first access to each of these pages will be for write. In a traditional cache, this access generates a miss, causing the copy of the page from the primary memory to the cache; however, the primary memory contains no valid data for this page, so this fetch action produces no useful effect. In our cache, these misses are not generated. As seen in Subsection 3.1, this improved fetch-on-miss strategy is implemented by extending the meaning of the dirty flags to invalid lines. The *Expand* command clears the validity and the dirty flags of the lines reserved for the new activation record, to signify that the primary memory contains no information for the corresponding pages. In this way, no fetch-on-miss will take place for these lines.

5.3. Miss ratio

The probability that a given data item is referenced is high if this data item is stored in the local activation record at the stack top or in the global activation record at the stack base, and is low if it is stored in a non-local activation record at an intermediate stack position. This aspect of program behaviour was exploited by Weicker (1984) in the design of the Dhrystone synthetic benchmark and by Tanembaum (1978) in his proposal for a machine architecture for the execution of structured programs, for instance. It has been confirmed by several experimental studies. Carter (1982) analysed the static properties of 89 Pascal programs that included compilers, assemblers, interpreters, text processors, numeric and graphic programs, for a total of over 95 000 lines of source code. The programs defined 2415 procedures and the mean number of procedures for each program was 27.13. In these programs, 40.71% of the variable accesses were local accesses, 52.69% were global accesses and only 6.60% were non-local accesses. Cook and Lee (1982) studied a sample of 264 Pascal programs for the Digital Equipment VAX 11/780 and PDP 11/45 computers. The programs consisted of 120 239 lines of code, and included over 3500 procedure and functions. In these programs, 52.6% of the variable accesses were local accesses, 45.4% were global accesses and 2.0% were non-local accesses.

Let us now denote the cache miss ratio by MR, and let MR_{GAR}, MR_{LAR}, and MR_{NLOC} be the components of MR corresponding to the misses originating from the accesses to the global, the local and the non-local activation records, i.e. $MR = MR_{GAR} + MR_{LAR} + MR_{NLOC}$. Let us suppose that, at any given time, both the global and the local activation records can be entirely contained in the cache, i.e. $\Delta_{GAR} + \Delta_{LAR} \leqslant N$ (the next subsection shows that this hypothesis is realistic). In this case, the pages of these two activation records share no cache line. If we hypothesize that in a short transient at the beginning of the program execution all the global pages become resident in the cache, after this transient we will have $MR_{GAR} = 0$ and $MR = MR_{LAR} + MR_{NLOC}$.

As seen in Section 4, when execution of a subprogram terminates, a *Shrink* command is issued which decreases the value of the top pointer. As results from relation (6), this change modifies of the position in the cache of all the high pages. Thus, the previous subprogram whose execution is now resumed will find no page of its own address space in the cache. This gives raise to a high miss rate, which rapidly diminishes while execution continues. Eventually, when the whole subprogram activation record has been loaded into the cache, we will have $MR_{LOC} = 0$ and $MR = MR_{NLOC}$. As seen above, the non-local references are a small fraction of the total. It follows that in the steady state the cache miss ratio will be low.

It should be noted that the above-mentioned form of cold start (Agarwal, 1989; Hennessy and Patterson, 1990) which takes place on subprogram termination will not be observed when execution of a new subprogram is started up. This is a consequence of the fact that, at the beginning of the execution of a given subprogram, the primary memory stores no valid information for the pages of the activation record inserted in the stack for this subprogram. As seen previously, the dirty flags allow us to take advantage of this aspect of the stack paradigm of memory references and generate no miss at the first access to each of these pages. This is in contrast to what happens in a traditional cache, where a high miss rate characterizes every change in the program locality of reference.

5.4. Cache size

Experimental studies indicate that the mean activation record size is moderate. Batson and Brundage (1977) analysed the static and dynamic properties of a sample of 34 Algol 60 programs for the Burroughs B5500 (static and dynamic program properties can differ (Knuth, 1971); in the static measurements, each data reference contributes one value, in the dynamic measurements a data reference that is executed several times, e.g. in a loop, contributes one value for each execution, and a reference which is never executed contributes no value). The memory requirements for the sample programs ranged from 162 to 89 976 48-bit words. The smallest program contained 1 Algol block, the largest 99 blocks, for a total of 280 blocks (34 global blocks and 246 inner

blocks). The B5500 Algol compiler reserved two segments for each block activation, one segment for the parameters and the local variables (excluding arrays) and the other for the local arrays. The measurements included the memory requirements of the segments of both types, and the total memory requirements of each block activation. Dynamically, the mean total memory requirement for each activation was 28.1 words (including arrays).

One could doubt the general validity of the Batson and Brundage data on account of the upper limit of 1023 words imposed by the Burroughs B5500 architecture on segment sizes. However, other experiments confirm that the mean memory requirements for activation record storage are low, and, consequently, the B5500 segment size limit should not be a real bias (Ditzel, 1980). For instance, Ditzel and McLellan (1982) measured the dynamic distribution of the activation record sizes in a sample of C programs consisting of all the standard UNIX commands, for a total of 53 197 lines of source code. They found that the size of 91.1% of the activation records was 31 32-bit words or less and the size of 94.5% of the activation records was 127 words or less.

However, in most cases, the global activation record is significantly larger than the inner activation records. This effect follows from the common programming practice to allocate the arrays at the global level. It has been quantified by Gehringer in his book (1982) on capability architectures and small objects. Let N_G and N_L denote the mean number of arrays per global and local activation record, respectively. By elaborating the data from Batson and Brundage, Gehringer found $N_G = 11$ and $N_L = 0.351$. Similar results were obtained by Brookes *et al.* (1982) who statically analysed a sample of 11 Pascal programs for scientific and non-scientific applications. The program sizes ranged from 225 to 8694 lines of source code. In the scientific programs, the arrays defined at the global level were 55% of the total, in the non-scientific programs the global arrays were 86% (in contrast, in the two samples, only 26 and 38% of the declared variables were global variables). (An indirect confirmation of the fact that most arrays are declared at the global level can be obtained by observing the lifetimes of the program entities. In the Batson and Brundage programs, the mean array lifetime was 6.7 s and the mean activation record lifetime was 32.7 ms.)

We can obtain a rough estimate of the mean sizes of the global and the local activation records by separating the memory requirements of arrays from those of the other activation record items. Let Δ_R denote the mean dynamic activation record size (excluding arrays) and Δ_A denote the mean dynamic array size. Then we have $\Delta_{GAR} = \Delta_R + N_G \cdot \Delta_A$ and $\Delta_{LAR} = \Delta_R + N_L \cdot \Delta_A$. In the Batson and Brundage programs, Δ_R was 17.9 words and Δ_A, 343.1 words. Thus, $\Delta_{GAR} = 3792$ words and $\Delta_{LAR} = 138.3$ words. However, the value of Δ_A was biased by a small number of very large arrays (the memory demand of the largest array was 65 792 words), as is confirmed by the small value of the median of the array size (16 words). In fact, 80% of the arrays were smaller than 50 words and 90% were smaller than 122 words (in these two cases, we have $\Delta_{GAR} = 567.9$ words and $\Delta_{GAR} = 1359.9$ words, respectively).

5.5. Utilization of the cache storage space

The considerations presented so far allow us to conclude that a cache of, for instance, 10 Kbyte, will be usually able to store both the local and the global activation records of the running program. In a few cases, the total size of these activation records will be substantially larger than the cache size, and in other few cases these activation records will be substantially smaller than the cache. However, our cache is always able to profitably use its own storage resources. In particular:

- if $\Delta_{GAR} + \Delta_{LAR} > N$, the mechanism of the shared cache lines makes it possible to store a high (low) page in the cache as long as the low (high) page mapping into the same line is not referenced.
- if $\Delta_{GAR} + \Delta_{LAR} < N$, the cache lines not used for the global and the local pages will be reserved for the pages of one or more non-local activation records stored at low stack positions, near the stack base (the motivation is that the lifetimes of these activation records are longer than the lifetimes of the activation records near the top).

5.6. Commands

Of course, the cache commands are a source of system performance degradation, in both terms of the space required to store them and the increase in the memory traffic due to the need to fetch them. A single *Initialize* command is required for each program, so the costs connected with this command are negligible. Let C denote the mean size of the subprogram code segments (excluding commands), and let D_E and D_S denote the memory size of the *Expand* and *Shrink* commands. Moreover, let ρ denote the increase of the code segment size due to the storage of these commands, $\rho = (D_E + D_S)/C$. As seen in Section 4, the command parameters are activation record sizes. We will hypothesize that one such size can be codified in two bytes. This hypothesis is realistic; for instance, in the Ditzel and McLellan's distributions (1982) the size of 99.3% of the activation records was 512 32-bit words or less (of course, it is always possible to express activation record sizes in terms of a multiple of 2^n bytes, so reducing the memory requirements of the parameters by n bits). If a command operation code can be codified in one byte, we obtain $D_E = 3$ bytes and $D_S = 4$ bytes. In the Batson and Brundage programs, C was 93.1 48-bit words statically, and 38.8 words dynamically. For these programs, we obtain $\rho = 1.3\%$ statically and 3% dynamically. Of course, the static data are significant for the memory requirements for code storage, and the dynamic data for the memory traffic generated by the program execution.

5.7. Strategies for the compiler

The activity of our cache is controlled by the software. This means that the compiler has to insert the cache commands at appropriate points of the machine code translation of the source program. Putting more burden on the compiler is a modern tendency on which several recent architectures rely, e.g. reduced instruction set computers (Bradlee *et al.*, 1991; Kane and Heinrich, 1992).

As seen in Section 4, the compiler must generate the *Initialize* command in the program initialization part. The command parameters are the size of the global activation record, that can be evaluated statically, and the page number of the stack base, that is available at run time in the stack pointer register.

A simple compiler strategy for the *Expand* and *Shrink* commands is to insert them in the machine code translation of each subprogram, in the subprogram prologue and epilogue, respectively. Thus, *Expand* will be executed as part of the actions performed when a subprogram is entered. This command has a single parameter, i.e. the size of the activation record of the new subprogram. In a number of languages, e.g. Pascal and C, this size is known statically. This is not the case for languages, such as Algol 68 and Ada (Ada is a registered trademark of the US Government, Ada Joint Program Office), featuring variables whose size can only be determined at run time, e.g. dynamic arrays (Ghezzi and Jazayeri, 1987). In these languages, the memory requirements of the activation record of a given subprogram are only known when this subprogram is activated. However, even in a language of this type, an important property is that the activation record sizes never vary during subprogram execution. This allows us to use the result of the size evaluation performed when a subprogram is activated not only for stack management purposes, but also in the *Expand* command to control the cache activity.

Finally, *Shrink* has two parameters, i.e. the size of the activation records of the terminating subprogram and of the subprogram whose execution is being resumed. The values of both these parameters are always computed at run time as part of the information needed to update the stack. We now require that the compiler uses the results of these computations in the *Shrink* command to manage the cache space.

A compiler directive may be provided, stating that the *Expand*, *Shrink* command pair should not be generated for a given subprogram. Consequently, no cache space will be allocated for this subprogram, every memory reference issued by the subprogram will be carried out in the primary memory and the cache will generate no miss on subprogram termination to restore the state of the previous subprogram. In this way, the programmer can take advantage of an explicit software control over the cache activities. In our cache, the activation record is the basic unit of storage space management. This contrasts with what happens in traditional caches, where the line is the unit of cache space management, and lines are allocated and made free automatically, on the occurrence of the cache misses.

Our cache uses the lines not reserved for the local and the global activation records to store the pages of the non-local activation records near the stack base. The compiler can modify this policy and use these lines for the pages of the non-local activation records near the top. This can be useful, for instance, if a subprogram generates a call to another subprogram with moderate memory requirements, e.g. a small library routine; by letting the activation record of the calling subprogram reside in the cache during execution the routine, we will save the cache misses required to restore the state of the caller on termination of the routine. To obtain this effect, the compiler will generate a single *Expand*, *Shrink* command pair, on the beginning and on termination of the execution of the first subprogram, to reserve and free the cache space required for the activation records of both this subprogram and the routine.

6. CONCLUDING REMARKS

We have approached the problem of reducing the memory bandwidth requirements of programs written in block-structured, high-level languages by means of a cache memory able to store portions of the stack used in the implementation of these languages. Our main goals have been:

- To decrease the complexity of the logic implementing the functionalities of cache space addressing and management.
- To increase the cache performance by taking advantage of a compile-time knowledge of salient aspects of the program memory behaviour.

We have proposed the architecture of a stack cache aimed at satisfying these requirements. In this architecture:

- The address space is partitioned in pages. The pages of the program stack area are classified into high and low, according to their position near the stack top and base, respectively. Usually, the high pages include all the pages of the local activation record, and the low pages include all the pages of the global activation record. Each cache line will be reserved to contain a high or a low page. In a small cache, a number of lines will be shared between the high and the low pages.
- Translation of a page number into the corresponding line number is carried out by means of a form of direct mapping (Hill, 1988), taking into account the classification (high or low) of the page being referenced.
- A set of cache commands implements a form of direct communication between the running program and the cache. The compiler will insert the commands at appropriate points in the program object code, to

transmit the cache information concerning the program structure and usage of memory resources.

We have obtained the following results:

- The direct mapping of memory addresses into cache addresses and the software control over the cache operations decrease the complexity of the cache management functionalities significantly. The resulting saving of cache logic will be of particular interest in an on-chip implementation of the cache with the processor.
- The selection, performed at the cache addressing level, of the stack pages that have a high probability of being referenced decreases the cache miss ratio. Measurement studies indicate that these pages belong to the local and the global activation records.
- A partial copy-back strategy reduces the memory traffic by not restoring the state of the primary memory when a line is selected for replacement, if this line contains a discarded page belonging to the activation record of a terminated subprogram. On the occurrence of a miss involving a line of this type, the processor will not have to wait for the copy-back to complete, for enhanced performance.
- A partial fetch-on-miss strategy decreases the number of line fetches by not loading the pages of the activation record of a new subprogram at the first reference to each of them.

Our cache architecture implements a form of software control over the cache activities. This is certainly not a new idea; it has been widely used in the implementation of data prefetching (Klaiber and Levy, 1991; Mowry *et al.*, 1992) and cache coherency protocols for multicache systems (Lopriore, 1989; Owicki and Agarwal, 1989), for instance. We believe this idea deserves fresh consideration in the implementation of a stack cache for block-structured programs.

ACKNOWLEDGEMENTS

The work described in this paper has been supported by the National Programme on "Information systems and parallel computing" of the Italian National Research Council, and by the Ministero dell'Universitá della Ricerca Scientifica Tecnologica, Italy.

REFERENCES

Agarwal, A. (1989) *Analysis of Cache Performance for Operating Systems and Multiprogramming*. Kluwer, Boston, MA.

Batson, A. P. and Brundage, R. E. (1977) Segment sizes and lifetimes in Algol 60 programs. *Commun. ACM*, **20,** 36–44.

Bradlee, D. G., Eggers, S. J. and Henry, R. R. (1991) The effect on RISC performance of register set size and structure versus code generation strategy. In *Proc. Eighteenth Annual Inter. Symp. on Computer Architecture*, pp. 330–339, Toronto, Canada.

Brookes, G. R., Wilson, I. R. and Addyman, A. M. (1982) A static analysis of Pascal program structures. *Software—Practice and Experience*, **12,** 959–963.

Carter, L. R. (1982) *An Analysis of Pascal Programs*. UMI Research Press, Ann Arbor, MI.

Cook, R. P. and Lee, I. (1982) A contextual analysis of Pascal programs. *Software—Practice and Experience*, **12,** 195–203.

Ditzel, D. R. (1980) Program measurements on a high-level language computer. *Computer*, **13**(8), 62–71.

Ditzel, D. R. and McLellan, H. R. (1982) Register allocation for free: the C Machine stack cache. In *Proc. Symp. on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, CA. In *Computer Architecture News*, **10**(2), 48–56.

Farrens, M. K. and Pleszkun, A. R. (1989) Improving performance of small on-chip instruction caches. In *Proc. Sixteenth Annual Int. Symp. on Computer Architecture*, Jerusalem, Israel. In *Computer Architecture News*, **17**(3), 234–241.

Gehringer, E. F. (1982) *Capability Architectures and Small Objects*. UMI Research Press, Ann Arbor, MI.

Ghezzi, C. and Jazayeri, M. (1987) *Programming Language Concepts*, 2nd Edition. John Wiley, New York, N.Y.

Hennessy, J. L. and Patterson, D. A. (1990) *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA.

Hill, M. D. (1988) A case for direct-mapped caches. *Computer*, **21**(12), 25–40.

Hwu, W. W. and Chang, P. P. (1989) Achieving high instruction cache performance with an optimizing compiler. In *Proc. Sixteenth Annual Int. Symp. on Computer Architecture*, Jerusalem, Israel. In *Computer Architecture News*, **17**(3), 242–251.

Kain, R. Y. (1989) *Computer Architecture Software and Hardware*. Prentice-Hall, Englewood Cliffs, NJ.

Kane, G. and Heinrich, J. (1992) *MIPS RISC Architecture*. Prentice-Hall, Englewood Cliffs, NJ.

Klaiber, A. C. and Levy, H. M. (1991) An architecture for software-controlled data prefetching. In *Proc. Eighteenth Annual Int. Symp. on Computer Architecture*, pp. 43–53, Toronto, Canada.

Knuth, D. E. (1971) An empirical study of FORTRAN programs. *Software—Practice and Experience*, **1,** 105–133.

Lopriore, L. (1989) Software-controlled cache coherence protocol for multicache systems. *Inf. Proc. Lett.*, **33**(3), 125–130.

Lopriore, L. (1993) A data cache for Prolog architectures. *Future Generation Computer Systems*, **9,** 219–234.

Matick, R. E. (1989) Functional cache chip for improved system performance. *IBM J. Res. Dev.*, **33,** 15–32.

Mowry, T. C., Lam, M. S. and Gupta, A. (1992) Design and evaluation of a compiler algorithm for prefetching. In *Proc. Fifth Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, Boston, MA. In *Computer Architecture News*, **20**(Special Issue), 62–73.

Owicki, S. and Agarwal, A. (1989) Evaluating the performance of software cache coherence. In *Proc. Third Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, Boston, MA. In *Computer Architecture News*, **17**(2), 230–242.

Pratt, T. W. (1984) *Programming Languages: Design and Implementation*, 2nd edn. Prentice-Hall, London.

Silberschatz, A., Peterson, J. L. and Galvin, P. (1991) *Operating System Concepts*, 3rd edn. Addison-Wesley, Reading, MA.

Slater, M., Editor (1992) *A Guide to RISC Microprocessors*. Academic Press, San Diego.

Smith, A. J. (1982) Cache memories. *Comp. Surv.*, **14,** 473–530.

Smith, J. E. and Goodman, J. R. (1985) Instruction cache replacement policies and organizations. *IEEE Trans. Comp.*, **C-34,** 234–241.

Stone, H. S. (1993) *High-Performance Computer Architecture*, 3rd edn. Addison-Wesley, Reading, MA.

Tanenbaum, A. S. (1978) Implications of structured programming for machine architecture. *Commun. ACM*, **21,** 237–246.

Watt, D. A. (1993) *Programming Language Processors*. Prentice-Hall, New York, NY.

Weicker, R. P. (1984) Dhrystone: a synthetic systems programming benchmark. *Commun. ACM*, **27,** 1015–1030.