# Code Partitioning in Decoupled Compilers

Kevin D. Rich and Matthew K. Farrens

University of California at Davis

**Abstract.** Decoupled access/execute architectures seek to maximize performance by dividing a given program into two separate instruction streams and executing the streams on independent cooperating processors. The instruction streams consist of those instructions involved in generating memory accesses (the *Access* stream) and those that consume the data (the *Execute* stream). If the processor running the access stream is able to get ahead of the execute stream, then dynamic pre-loading of operands will occur and the penalty due to long latency operations (such as memory accesses) will be reduced or eliminated.
Although these architectures have been around for many years, the performance analyses performed have been incomplete for want of a compiler. Very little has been published on how to construct a compiler for such an architecture. In this paper we describe the partitioning method employed in *Daecomp*, a compiler for decoupled access/execute processors.

## 1 Introduction

Program execution can be viewed as a two-part process — the moving of data to and from memory and the performing of some operation on that data. Conceptually, these two steps can be represented by two cooperating processes, the *memory access* process and the *computation* (or *execute*) process.

A decoupled architecture seeks to achieve high performance by running these two processes on separate cooperating processors, allowing out-of-order execution between the two instruction streams. The processors both traverse the same dynamic flow graph, though not necessarily at the same pace. To allow the processors to execute different portions of the graph, architecturally visible queues are employed to buffer information produced by one process for consumption by the other. If the access process can run sufficiently ahead of the execute process on this flow graph then it will dynamically preload the operands consumed by the execute process and hide the latency of memory access operations. If this occurs it is said that access process has *slipped* with respect to the execute process, or that *slip* has been achieved.

The execution of the two processes on separate processors provides a simple mechanism for supporting limited out-of-order execution, and the ability to issue more than one instruction per cycle. Because of its simplicity and potential ability to tolerate long memory latencies, there is a continuing interest in decoupled architectures [8, 9, 10, 3, 12]. In particular, decoupled architectures

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 1008–1017, 2000.

may be of great interest in the growing field of power-conscious processor design because the simplified method of exploiting ILP does not require many of the large, power-hungry circuits needed by superscalar designs.

While decoupled processing has existed in various incarnations for years [7, 10, 1, 9] there has been very little published on the compilation techniques necessary. The most fundamental compilation issue is how the instructions/operations will be allocated or assigned to the access and execute processors. This process is referred to as the *partitioning* of the code. In the rest of this paper we describe the partitioning scheme employed by Daecomp, an ANSI-C compiler we developed to allow for a more complete analysis of decoupled processing. An example of actual compiler output will be shown, as will some simulation results obtained using Daecomp-produced code.

## 2 Background

Despite the appearance of decoupled architectures in the literature for many years, very little information is available regarding the techniques necessary for a decoupled compiler. At least two functioning decoupled compilers exist that we are aware of, but both were for commercial products (the ZS-1 [7] and the ACRI-1 [9]) and therefore details of the compiler construction have not been published.

The research most closely related to the work we are presenting here was performed by Topham et al. [8] in which they investigate source-code level transformations that can be made for the ACRI-1 with the goal of reducing the frequency of AP-EP synchronization. Our work focuses on the lower-level partitioning task performed by the compiler.

## 3 Processor Model

The two cooperating instruction streams produced by a decoupled compiler will execute on two separate processing elements, which communicate in a message passing manner via architecturally visible queues. A block diagram of the target architecture is shown in Figure 1. Memory addresses are sent to memory via the Load and Store Address Queues (LAQ,SAQ), and memory operands are received or sent via the Load and Store Data Queues (LDQ,SDQ). In addition to the LAQ-LDQ and SAQ-SDQ queue pairs, each processor in this model has a complete set of *alternate* memory queues. These queues allow the processor to perform loads and stores on its own behalf (*self-loads* and *self-stores*). There are also Copy Queues (to allow data transfer between the processors) and Branch Queues (to provide control flow synchronization).

Use of these queues allows instructions from the two instruction streams to slip with respect to one other, providing dynamic scheduling and out-of-order issue capabilities without requiring the architectural complexity of a superscalar processors instruction window. Decoupling seeks to tolerate memory latency via this dynamic scheduling, in particular via the early issue of memory operations.

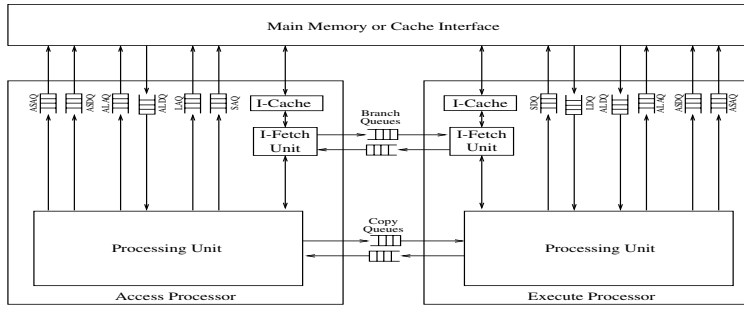



**Fig. 1.** Decoupled Access/Execute Processor - Conceptual Diagram

## 4 The Compiler

Daecomp is based on *cbend*, a multi-threaded compiler for the Concurro architecture produced by Bernard K. Gunther. The multi-threaded aspects of cbend were stripped out and the code specific to decoupled compiling was added. In addition, cbend's register allocator and code emission routines underwent modifications, primarily to deal with the use of queues.

Aside from partitioning the code, perhaps the biggest challenge encountered during compiler construction was handling function calls efficiently and correctly. Since the compiler is responsible for parameter set-up on the stack and/or in the parameter registers (each processor has its own register file), one major design decision is whether each processor will have a private copy of the run-time stack or if a single stack will be shared.

The decision impacts how function calls are managed, requiring different protocols for parameter passing and different considerations for local (i.e., stack-based) variable management. Passing parameters to a function under a single shared stack model is simpler than under a private stack model, but since the stack based variables are a shared resource there are a variety of potential performance ramifications. For performance reasons a dual-stack model was initially explored — unfortunately it proved to have a fatal flaw related to passing pointers to pointers so the partitioning technique detailed here assumes a single, shared run-time stack.

## 5 Code Partitioning

The task of the compiler is to partition a directed acyclic graph into two separate, cooperating, decoupled instruction streams based on def-use information. The graph consists of *nodes*, which represent the instructions/operations to be performed, and *edges*, which convey the dependencies between nodes. The goal is to produce a partitioning that reasonably balances the processor workload and

makes it possible for the two instruction streams to slip with respect to each other.

In the partitioning technique employed by Daecomp, each operation is assigned to a processor based on the class of operation (e.g., address generation, function parameter set-up, etc.) to which it belongs.

To start the marking process operations like loads and stores are designated as *anchors*. Anchor nodes are either *sinks* or *sources* of expressions. Sink nodes represent operations (e.g., stores) upon which no other operation directly depends, while source nodes represent operations which do not depend directly on any other operation (e.g., dequeuing a data value). Once the anchors are identified, the graph is again traversed, this time assigning (or *marking*[1]) instructions which depend on an anchor (or on which an anchor depends) to the same processor as the anchor. Interprocessor data dependencies are handled by inserting copy instructions from one processor to the other where necessary.

Once the partitioning (detailed below) is completed the compiler has created two graphs, one for each processor. Register allocation and code emission are performed on each graph resulting in the decoupled object code file. The five-step partitioning process will now be detailed.

**Step 1: Anchor Return Value Usage**

To minimize inter-processor copies it is desirable to anchor a function's return value calculation on the processor which is going to use the value. Unfortunately, a function may be called from several different locations within the program (or programs, in the case of a library routine) and the return value may be needed on different processors depending upon the calling location. Additionally, it is unlikely the compiler will have any information regarding the caller(s) of the function that it is compiling.

Therefore, it must be determined a priori which is the *return value processor.* The node which puts the return value into the return value register is marked for the return value processor and serves as an anchor. The compiler currently requires all return values to reside on the EP[2].

**Step 2: Split Loads and Stores**

The splitting of memory accesses into address and data portions lies at the heart of decoupled processing. Load operations are split into two parts: the operation that enqueues the load address onto the load queue, and the operation that dequeues the loaded data from the load data queue. Stores are similarly split into the operation which enqueues the store address and the operation which enqueues the store data.

[1] A node which is *marked* has been assigned to a processor. A node which has not yet been assigned to a processor is *unmarked.*

[2] In the future, the compiler will be modified to handle return values more intelligently. For a function which does not return a pointer it will assume that the value computed is needed by the EP and thus the EP will be the return value processor. Pointer valued functions are most likely computing a value related to an address calculation, so the AP will be the return value processor if the return value is a pointer.

| UNIPROCESSOR | ⇒ | ACCESS PROCESSOR | EXECUTE PROCESSOR |
|---|---|---|---|
| load rA, rB, 8 | | add LAQ, rB, 8 | move rA, LDQ |
| store rX, rY, 8 | | add SAQ, rX, 8 | move SDQ, rY |

As indicated in the above example, the node which enqueues the load or store address is assigned to the AP and the node which enqueues or dequeues the data is marked for the EP; these nodes serve as anchors. For the load operation, the node for the instruction which dequeues from the LDQ is a source node and the node which enqueues the load address on the LAQ is a sink. For the store operation, both the node for the instruction which enqueues the address on the SDQ and the node which enqueues the data on the SDQ are sinks.

It is important to note that while the processors may be identical, the EP cannot be allowed to enqueue addresses for data which the AP can access (e.g., global data or data on the AP's stack), because there is the potential for the violation of RAW, WAR, or WAW dependencies. So *functionally* speaking such *reverse decoupled* loads and stores are not permitted. In the shared stack model these restrictions result in the EP not be being permitted to use its alternate memory queues.

**Step 3: Propagate Load/Store Processor Markings**

Propagation of markings entails starting at each of the anchors operations and traversing the (sub)graph rooted at that anchor, assigning all unmarked data-dependent nodes in the graph to the same processor as the anchor. The propagation occurs in two directions, down from a sink and up from a source. When propagating the markings down from a sink, the marking propagates down through its children to all of its descendants. When propagating the marking up from a source, the marking propagates up through its parents to all of its ancestors.

If during the traversal a node is encountered that is marked for the other processor, a copy is necessary in order to communicate the value from one processor to the other. In the below example of an AP-to-EP copy, AP line 2 enqueues the value on the copy queue, and EP line 1 dequeues the value from the copy queue.

| | ACCESS PROCESSOR | EXECUTE PROCESSOR |
|---|---|---|
| 1: | add rA, rB, 8 | sub rY, CPQ, rX |
| 2: | move CPQ, rA | |

The compiler attempts to avoid EP-to-AP copy operations as they introduce slip-reducing dependencies. The fact that a copy is required means that the value is needed by both processors and thus the computing expression is a common sub-expression (CSE). If the CSE is inexpensive to compute, *common sub-expression replication* may be employed (in which both processors perform the calculation of the sub-expression). If the CSE is too expensive to replicate, but cheap enough that moving it to the AP would not lead to grossly unbalanced code, it is moved to the AP (it is *stolen*) and an AP-to-EP copy is inserted. Allowing the AP to steal the sub-expression eliminates the need for the slip-reducing

EP-to-AP copy. If this *common sub-expression stealing* would result in code balancing problems, or is otherwise infeasible, an EP-to-AP copy is inserted. The compiler makes these decisions by estimating the computation cost of the sub-expression and comparing it to threshold values[3]. Copies related to load and store operations may also be eliminated by converting standard decoupled loads or stores into self-loads or self-stores on the access processor.

If an EP-to-AP copy is unavoidable, the compiler performs code scheduling in order to minimize the impact of the copy. The EP's enqueuing operation is placed as early as possible in its instruction stream, and the AP's dequeuing operation is placed as late as possible in its instruction stream. This technique is used extensively with function parameters.

**Step 4: Branch Splitting**

After marking propagation, conditional branch operations are split into their two cooperating counterparts, the *external branch* operation and the *branch from queue* operation. An external branch is a conventional branch which also writes the branch decision (taken/not taken) to the processor's outgoing branch queue, while a branch from queue (`bfq`) is a conditional branch that is taken or not based on the value at the head of the processor's incoming branch queue. This mechanism allows the processors to traverse the flow graph in identical fashion by easily synchronizing control flow decisions. In the below example the `x` in the `brxnz` indicates an external branch.

| UNIPROCESSOR | ⇒ | ACCESS PROCESSOR | EXECUTE PROCESSOR |
|---|---|---|---|
| `cmpne rC, rA, rB` | | `cmpne rC, rA, rB` | |
| `brnz rC, @L1` | | `brxnz rC, @L1` | `bfq @L1` |

This splitting is only performed on conditional branches. Unconditional flow control operations (e.g., jumps, function calls) are simply duplicated on each of the processors and do not need to be split.

The compiler must determine which processor is to receive each of the two branch operation components. There are several issues to be considered when making this decision — for example, if the `bfq` is put on the AP then a slip reducing AP-on-EP dependency is introduced. In order to avoid such dependencies, the compiler makes every attempt to put the comparison and the external branch on the AP and the `bfq` on the EP. Forcing the comparison to be on the AP may, however, significantly impact code balance between the AP and EP. For example, functions which approximate solutions to equations often iterate until two successive approximations are within a pre-specified limit. In this case the EP is the natural choice to perform the value computations, and moving these computations to the AP would leave the EP with little or nothing to do, negatively impacting performance. Therefore, the compiler takes into account both existing processor markings and the cost of the sub-expression which performs the comparison when determining the branch assignments.

[3] If an AP to EP copy is required then the only consideration is code expansion due to the copy operations. In this case replication of the CSE is employed only if the CSE consists of a small number of low-latency instructions.

**Step 5: Propagate All Markings**

At this point in the process the entire graph has been traversed and node markings have been propagated to the ancestors and descendants of the anchor nodes. The final step is to traverse the graph once more with node markings being propagated from all nodes. As described previously, copies are inserted where necessary and common sub-expressions may be replicated. The propagation continues as long as it results in changes to the graph (by copy insertion, processor markings, or common sub-expression replication). Once this process terminates there are no unmarked nodes and the two graphs have been extracted.

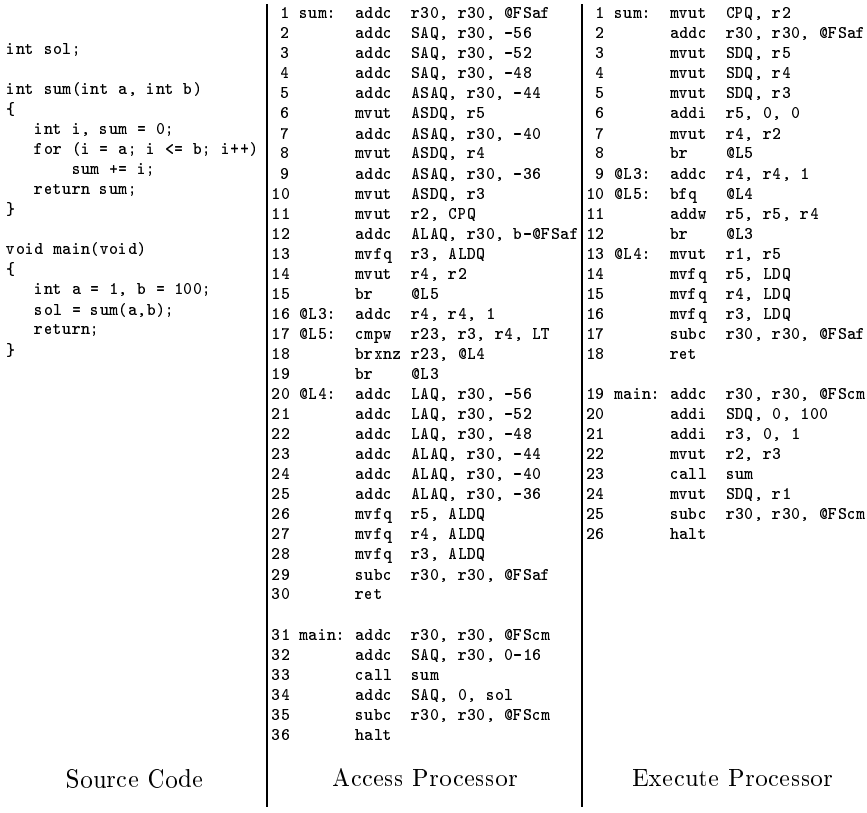
## 6 Example Compiler Output

This section presents the compiler output for a simple source file which computes the sum of a sequence of integers (shown in left-most column of Figure 2). This program was chosen because it includes examples of decoupled loads and stores, self loads and stores, copies, external branches, and branch from queue operations. Figure 2 also shows the decoupled access/execute code generated by the compiler. The labels of the form `@FSxx` are stack frame sizes and are simply used to make adjustments to the stack pointer (`r30`). `r1` is the return value register. A single parameter register was used in order to illustrate parameter passing both in a register and on the stack. Relevant assembly language explanations are given in Table 1.

| `Opcode` | Description |
|---|---|
| `mvfq dest, queue` | Move from queue: $dest$ = value at head of $queue$ |
| `brnz src1, label` | Branch not zero: $if(src1)\{PC = label\}$ |
| `brxnz src1, label` | Branch external not zero: $if(src1)\{PC = label\}$, send results of branch to branch queue |
| `bfq label` | Branch from queue: $if(value\ on\ branch\ queue)\{PC = label\}$ |

**Table 1.** Selected Assembly Language Opcodes

Looking at Figure 2 one can see an example of a standard decoupled load on line 14 on the AP (AP.14) and line 21 on the EP (EP.21). AP.2 and EP.3 are the two halves of a standard decoupled store. An example of a slip-reducing copy is on lines EP.1 and AP.11; this particular copy is used to send the parameter passed in register 2 on the EP to the AP. Note that the slip reducing effect is mitigated by placing the producer operation before the SDQ accesses (EP.3 - EP.5) which save temporary registers for the EP, and the consumer operation after the corresponding SAQ, ASAQ, and ASDQ accesses (AP.2-AP.10) which assist the EP in saving its temporaries, and save the AP temporaries. EP.21 and EP.22 set up the first parameter to `sum()` in `r2`. AP.32 and EP.20 store the second parameter to `sum()` on the stack. AP.18 is an external branch and EP.10 is the corresponding branch from queue operation.



```
int sol;

int sum(int a, int b)
{
   int i, sum = 0;
   for (i = a; i <= b; i++)
       sum += i;
   return sum;
}

void main(void)
{
   int a = 1, b = 100;
   sol = sum(a,b);
   return;
}
```

Source Code

```
1 sum:  addc  r30, r30, @FSaf
2       addc  SAQ, r30, -56
3       addc  SAQ, r30, -52
4       addc  SAQ, r30, -48
5       addc  ASAQ, r30, -44
6       mvut  ASDQ, r5
7       addc  ASAQ, r30, -40
8       mvut  ASDQ, r4
9       addc  ASAQ, r30, -36
10      mvut  ASDQ, r3
11      mvut  r2, CPQ
12      addc  ALAQ, r30, b-@FSaf
13      mvfq  r3, ALDQ
14      mvut  r4, r2
15      br    @L5
16 @L3: addc  r4, r4, 1
17 @L5: cmpw  r23, r3, r4, LT
18      brxnz r23, @L4
19      br    @L3
20 @L4: addc  LAQ, r30, -56
21      addc  LAQ, r30, -52
22      addc  LAQ, r30, -48
23      addc  ALAQ, r30, -44
24      addc  ALAQ, r30, -40
25      addc  ALAQ, r30, -36
26      mvfq  r5, ALDQ
27      mvfq  r4, ALDQ
28      mvfq  r3, ALDQ
29      subc  r30, r30, @FSaf
30      ret

31 main: addc  r30, r30, @FScm
32       addc  SAQ, r30, 0-16
33       call  sum
34       addc  SAQ, 0, sol
35       subc  r30, r30, @FScm
36       halt
```

Access Processor

```
1 sum:  mvut  CPQ, r2
2       addc  r30, r30, @FSaf
3       mvut  SDQ, r5
4       mvut  SDQ, r4
5       mvut  SDQ, r3
6       addi  r5, 0, 0
7       mvut  r4, r2
8       br    @L5
9 @L3:  addc  r4, r4, 1
10 @L5: bfq   @L4
11      addw  r5, r5, r4
12      br    @L3
13 @L4: mvut  r1, r5
14      mvfq  r5, LDQ
15      mvfq  r4, LDQ
16      mvfq  r3, LDQ
17      subc  r30, r30, @FSaf
18      ret

19 main: addc  r30, r30, @FScm
20       addi  SDQ, 0, 100
21       addi  r3, 0, 1
22       mvut  r2, r3
23       call  sum
24       mvut  SDQ, r1
25       subc  r30, r30, @FScm
26       halt
```

Execute Processor

**Fig. 2.** Source and Assembly Code for Compilation Example

## 7 Results

The book *Numerical Recipes in C* contains a wide variety of algorithms, eight of which were selected as a representative sample [5]. These benchmarks have between 82 and 163 lines of source code which result in actual instructions in the executable (i.e., no comments or variable declarations are counted).

The decoupled simulator (*Decsim*) is written in C and accepts a simple object code format containing tuples of `<OPCODE OP1 OP2 OP3>`. The simulator models individual processors that are simple, single-issue, in-order execution, RISC style processors. They employ a five-stage pipeline with hardware interlocks and data forwarding. Each processor has 32 registers and assumes a perfect instruction cache. Infinite depth queues are used to make the results independent of any queue resource constraints. The memory latency of 18 cycles for the first word and 2 cycles for each subsequent word in a memory line/block was selected based on current memory technology. Decsim can operate in either a decoupled or uniprocessor mode. The uniprocessor mode does not use queues, instead re-

lying on standard load and store operations to communicate with memory. In the uniprocessor mode a data cache is employed; no data cache is used in the decoupled mode.

Figure 3 shows the speed-up achieved by running the benchmarks on the two-processor decoupled architecture vs. a uniprocessor architecture with an 8K-byte cache[4] at memory latency of 20 cycles. With the 20 cycle memory the average speed-up is only 1.06, with four of the benchmarks actually running slower on decoupled processor. In all four of poor performing benchmarks the AP spends over 50% of its cycles stalled on either an empty branch or copy queue. As a basis for comparison the original fourteen Livermore Loops were compiled and simulated showing considerable speed-up (2.05 on average) corroborating the results from the previous studies of decoupling using these benchmarks [11, 2].

Intuitively, a speed-up of less than 1.0 seems unlikely since the decoupled processor enjoys a 2:1 advantage in raw processor resources. However, poor decoupled performance can occur if the decoupled processor is unable to achieve slip and the AP experiences the full memory latency on each memory access, while good uniprocessor performance can result if the cache hit rate is high. If these two events occur for the same benchmark the decoupled processor would run significantly slower than the uniprocessor. Investigation into the run-time behavior indicates that the poor performance is attributable to control dependencies and copy operations significantly limiting the slip — research is ongoing on techniques to address these issues. Daecomp was used to perform many other experiments, the results of which are available in [6].

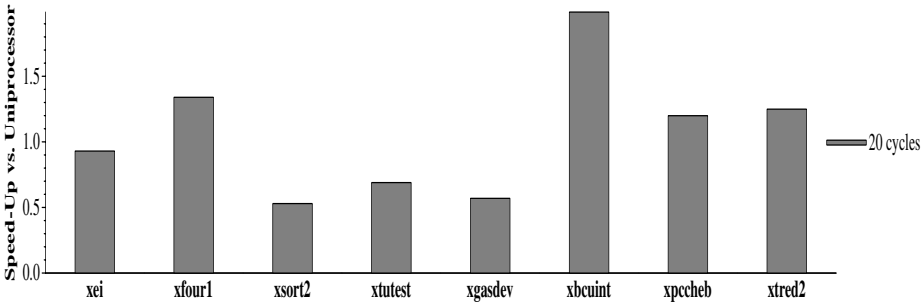



**Fig. 3.** Speed-Up Comparison with 20 Cycle Memory Latency — NRC

[4] If a typical cache size were used with the benchmarks in question, the cache would like suffer only compulsory/first reference misses, and conflict and capacity misses would likely not be an issue. Therefore, a small cache was used in order to keep it on scale with the working sets of the benchmarks. The cache selected resulted in overall hit rate of 97%.

## 8 Conclusion

To fully evaluate any architecture a compiler is needed, for this reason the decoupled access/execute compiler Daecomp was constructed. The primary role of a a decoupled compiler is to partition the instructions into two cooperating instruction streams. Daecomp implements a five-step partitioning process to identify the access and execute instruction streams. The results of some of these studies performed with Daecomp confirmed published result obtained using small, hand-compiled benchmarks [11, 1, 4, 2]. However, using larger, more varied benchmarks revealed that many of these earlier conclusions were erroneous, underscoring the importance of constructing a compiler. Further work is planned to determine if the architectural model can be modified to permit speculative execution, or to employ data caches.

## References

[1] Ali Berrached, Lee D. Coraor, and Paul T. Hulina. A decoupled access/execute architecture for efficient access of structured data. In *Proceedings of the 26th Annual Hawaii International Conference on System Sciences*, pages 438–447, 1993.

[2] Jian-tu Hsieh, Andrew R. Pleszkun, and James R. Goodman. Performance evaluation of the PIPE computer architecture. Technical Report 566, University of Wisconsin - Madison, November 1984.

[3] G.P. Jones and N.P. Topham. A comparison of data prefetching on an access decoupled and superscalar machine. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, 1997.

[4] William Mangione-Smith, Santosh G. Abraham, and Edward S. Davidson. The effect of memory latency and fine-grain parallelism on Astronautics ZS-1 performance. In *Proceedings of the 23rd Annual Hawaii International Conference on System Sciences*, pages 288–296, 1990.

[5] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brain P. Flannery. *Numerical Recipes in C*. Cambridge University Press, 2 edition, 1996.

[6] Kevin D. Rich. *Compiler Techniques for Evaluating and Extending Decoupled Architectures*. PhD thesis, University of California at Davis, 2000.

[7] James E. Smith. Dynamic instruction scheduling and the Astronautics ZS-1. *IEEE Computer*, 22(7):21–35, July 1989.

[8] Nigel Topham, Alasdair Rawsthorne, Callum McLean, Muriel Mewissen, and Peter Bird. Compiling and optimizing for decoupled architectures. In *Proceedings of the 1995 ACM/IEEE Supercomputing Conference*, 1995.

[9] N.P. Topham and K. McDougall. Performance of the decoupled ACRI-1 architecture: the Perfect Club. In *Proceedings of High Performance Computing - Europe*, 1995.

[10] Gary S. Tyson. *Evaluation of a Scalable Decoupled Microprocessor Design*. PhD thesis, University of California at Davis, 1997.

[11] Honesty Cheng Young. Evaluation of a decoupled computer architecture and the design of a vector extension. Technical Report 603, University of Wisconsin-Madison, 1985.

[12] Yinong Zhang and George B. Adams III. Performance modeling and code partitioning for the DS architecture. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, 1998.