Supporting Mixed Real-Time Workloads in Multithreaded Processors with Segmented Instruction Caches

Patrick Crowley
Applied Research Laboratory
Department of Computer Science & Engineering
Washington University
St. Louis, MO 63130-4899
pcrowley@cse.wustl.edu

Abstract

Current network processors (NPs) are being successfully deployed in networking systems and a host of other surprising application areas. This broad deployment is at once strengthened by the high computational throughput of these devices and hindered by program size limitations in the embedded processors providing that throughput. In this paper, we propose the use of segmented instruction caches *to remove program size restrictions for those programs that do not have real-time requirements without sacrificing the ability to provide guaranteed instruction delivery. In a segmented instruction cache, each thread is allocated a cache segment, or region, and any conflicts are restricted to within that segment. Several segment sizing strategies are introduced and evaluated, and profile-driven code scheduling is shown to often be effective at reducing miss rates for non-real-time code by between 10% and 60% over a set of benchmark programs.*

1. Introduction

Program size limitations are a hindrance in current network processors (NPs). NPs, such as the Intel IXP [8] network processor family, consists of two types of on-chip programmable microprocessors: a *control* processor, which is a standard embedded microprocessor (the ARM-based XScale), and multiple *data* processors which are microprocessors designed to efficiently support networking and communications tasks. The control processor is used to handle device management, error handling, and other so-called *slow-path* functionality. The data processors implement the packet processing tasks that are applied to most packets, i.e. the *fast-path* functionality, and are, therefore, the key to high-performance. In current NPs, each data processor executes a program stored in a private on-chip control store that is organized as a RAM. This fixed-size control store limits program size (e.g., the IXP2800 control store has 4K entries), which consequently both limits the variety of applications implementable on the NP and complicates the programming model. In this paper, we describe and evaluate a technique that removes this limitation without violating any of the tacit NP requirements that inspired the limitation in the first place.

Two questions arise immediately: do NP data processors need to execute large programs, and, if so, why are instruction caches not used? The answer to the first question can be seen in the large number of application areas that NPs are being deployed: traditional protocol and network processing systems such as wired and wireless routers, gateways, and switches; non-traditional networking tasks such as web-server load balancing, virus and worm detection and prevention, denial-of-service prevention systems; and other applications including cell-phone basestations, video distribution and streaming, and storage systems. This broad deployment is driven by the considerable processing and I/O bandwidths available on NPs, which are organized as heterogeneous chip-multiprocessors and therefore better able to exploit coarse-grain parallelism than are general-purpose processors. On the other hand, further deployment is hindered by program size limitations and the associated increase in software engineering costs.

As for the second question, while instruction caches are used in all general-purpose processors and most embedded processors, NPs have at least two characteristics that complicate their use. First, networking

systems are typically engineered to handle worst-case conditions [10]. This leads to real-time requirements for programs executing on NPs that are designed to meet worst-case specifications. Caches, on the other hand, are used to speed up average-case performance (and, in the worst-case when all references miss, can actually lower performance). Since caches rely on locality, and locality may not be present under worst-case conditions, their inclusion cannot always be justified if a system is designed for worst-case conditions. Secondly, NP data processors typically feature software-controlled, hardware-supported, coarse-grained multithreading. The presence of multithreading complicates instruction delivery in a processor, particularly while trying to provision a system for a certain performance target.

In this paper, we propose the use of a *segmented instruction cache* along with profile-driven code scheduling to provide flexible instruction delivery to both real-time and non-real-time (i.e., best-effort) programs, even when those programs are executing on the same multithreaded processor. Our proposed segmented instruction cache partitions the instruction cache (via a thread-specific address mapping function) such that each thread is allocated a fixed number of blocks in the cache and threads cannot conflict with one another. In this way, real-time programs can be tailored to fit into their assigned slots completely to avoid misses while best effort code, which can tolerate misses, can map larger, more complex control flow structures into their own segments without impairing the instruction delivery of other threads. To improve the performance of best-effort code, profile-driven code scheduling is used to reduce conflict misses. Profiling is also used to explore a variety of program-specific segment sizing strategies. The proposed design is evaluated on a selection of programs running on both direct-mapped and set-associative segmented instruction caches.

The remainder of this paper is organized as follows. Section 2 describes how programs are loaded and executed in NP data processors, and describes how a standard instruction cache might be used. Section 3 introduces the segmented instruction cache. An experimental evaluation of the proposal is presented in Section 4 and considers several benchmark programs, segment sizing, and reducing miss rates with profile-driven code scheduling and set-associativity. Section 5 presents related work; the paper ends with conclusions and future work in Section 6.

2. Instruction Delivery in NP Data Processors

Network processors, like most embedded processors and unlike most general-purpose processors, have rich compile-time knowledge of which programs and threads will be active at runtime. In fact, on the Intel IXP the source code for all the programs in all the threads to be run on a data processor is compiled together to form a single binary instruction image which is loaded by the control processor into the data processor's control store when the system boots. In the following sections, we discuss how the the program is mapped into the control store, and how a cache might be used in place of a RAM.

2.1. Fixed Size Control Store

Fixed size control stores are essentially random access memories (RAMs) containing instructions. Figure 1 illustrates an example with 3 programs, totaling 21 instructions in length, and a RAM with 18 memory locations. In the example, only the first two programs can fit; the third must be mapped to another data processor with enough available control store entries. Note that this information is known at compile time: when the programmer compiles her program, it will be known that all three programs cannot fit.

Since the control store is fixed, the total size of threads allocated to a single data processor has a hard upper bound. One consequence is that some programs are simply too big to fit in a single control store. In this case, the programmer must reduce the program size via optimization (if it is possible and it represents an acceptable performance tradeoff) or decompose the program into fragments to be mapped to multiple processors.

Of course, decomposing a task into balanced steps and mapping those tasks to multiple processors is a form of software-pipelining; this can be, and indeed is, done as a performance enhancement even when the control store size is not a consideration. However, while certain algorithms and programs lend themselves to balanced pipeline implementations, the efficient pipelining of general-purpose code is difficult. Thus, a method for flexible instruction delivery can alleviate the need to fragment a program when it is inconvenient or unnecessary to do so.

2.2. Using a Cache as a Fixed Size Control Store

As previously mentioned, a cache can alleviate program size restrictions. Furthermore, a cache can be

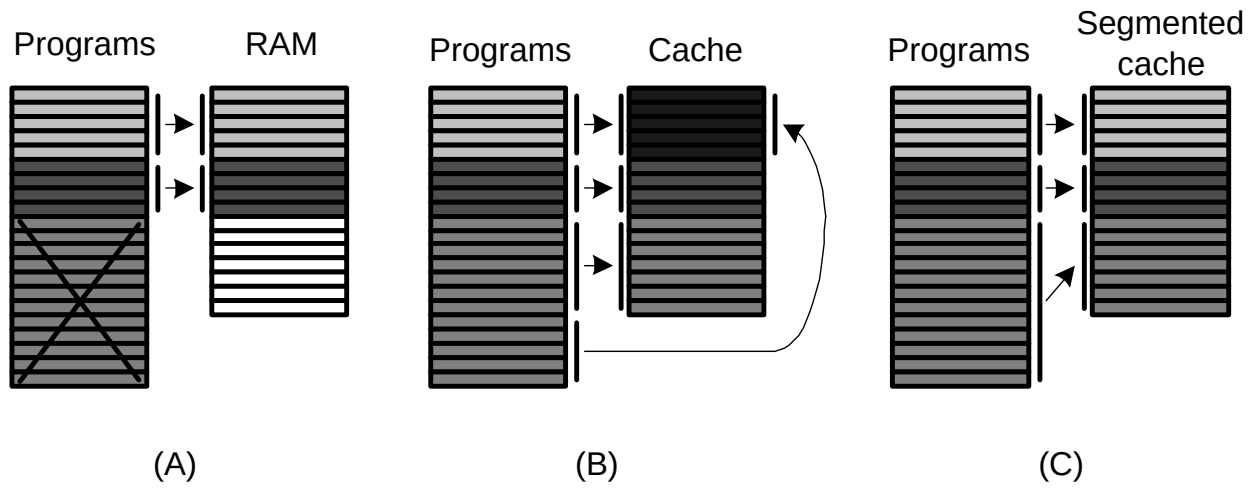

Figure 1. Illustrations mapping three programs from three distinct threads into (A) a fixed-size RAM, (B) a cache, and (C) a segmented cache, all of which contain 16 entries. The fixed size RAM cannot accommodate the third program. The cache can, but the mapping causes conflicts between the third and first programs. The segmented cache restricts conflicts to the segment assigned to program three.

used as a fixed size control store provided that the aggregate program size does not exceed the cache capacity and that code is laid out contiguously at the start of the address space. Given a cache with N entries, or sets, and an address a, then a mod N yields the entry address to which a maps. Thus, if the total number of instructions is less than N, then each will map to its own set. Given this scenario, the cache can be accessed without incurring misses as follows.

First, it is necessary to pre-load the cache with instructions as is done in the fixed control store case. Now, consider the three possible sources of uniprocessor cache misses [7]: compulsory, capacity, and conflict. Given that the program fits in the cache: pre-loading avoids compulsory misses, there will be no capacity misses, and conflict misses can be avoided by laying out the code sequentially in memory without any gaps. In this way, a cache can be used to implement a fixed size RAM. Note that no change is required in the programming model or compiler structure. However, a cache does involve some overhead in both access time and size; we return to these topics in Section 3.2.

Now, suppose one of our threads is not involved in providing stable service under worst-case conditions. In this case, our program can tolerate some instruction cache misses during execution. However, as illustrated in the second part of Figure 1, if the program size exceeds the contiguous unused portion of the cache, it will conflict with another thread (which may not be able to tolerate misses).

One way around this is to use cache line pinning to keep high-priority code from being evicted. When a cache line is pinned, it will not be replaced on a miss. This works well to keep important code in place, but always causes misses in the lower-priority conflicting code. There are other approaches to consider, and we discuss these in Section 5. As an alternative that does not require these unavoidable conflicts, we propose the segmented instruction cache.

3. Segmented Instruction Cache

In a segmented instruction cache, each thread is assigned a segment, and this provides two key benefits beyond the unrestricted program sizes afforded by caches. First, real-time programs that cannot tolerate misses can be allocated segments equal to their program size; this is equivalent to fixed control store case. The second benefit is that threads are insulated from one another and cannot conflict; most notably, real-time segments cannot be disturbed by other threads. This situation is illustrated in the third part of Figure 1.

To implement segments, we modify the cache mapping function to include a thread-specific segment size and offset. In other words, rather than having a constant N entries, the cache is seen to have $N(t)$ entries by thread t, and $offset(t)$ indicates the position of the first set in the segment. Thus, the address mapping

function becomes:

$$(a \bmod N(t)) + offset(t), \quad (1)$$

where a is the address and $N(t)$ and $offset(t)$ are thread t's segment size and offset within the cache, respectively.

Note that there is, again, no change in programming model for a real-time program. The program gets compiled, the required segment size is equal to the program size and thus is known. The program must be mapped to a data processor with a sufficiently large segment available. In fact, one would probably map all real-time programs on a given data processor to the same segment, and only assign unique segments to non-real-time code (since, as we will see, inter-thread conflicts can be problematic).

The data processor can now support arbitrarily large non-real-time programs. For such programs, however, we must determine a segment size.

3.1. Segment Sizing Strategies

When choosing a segment size, we first determine if a choice is available. It may be the case that other considerations dictate that a given non-real-time program must run in a particular amount of available space (e.g., whatever remains once the real-time code has been allocated). In this case there is no decision.

On the other hand, there may be some choice in the size of the allocation; in some fashion, a number of blocks needs to be determined. For example, there may be multiple non-real-time programs to be mapped to a particular data processor and appropriate segment sizes must be chosen. Accordingly, consider the following three segment sizing strategies.

- **program**. Choose a segment size equal to the number of instructions in the program. No possible execution will result in a miss. This is the strategy used for real-time code.
- **profile**. Choose a segment size equal to the number of unique references seen during a profiled execution run. In this case, misses will only occur when program execution takes place outside the profiled paths.
- **locality**. Choose a segment size equal to some fraction of the unique references seen during profiling. It is frequently the case that a small fraction of unique references account for a large majority of the dynamic references. In this case, misses are possible but will be few if the profiled data is representative to real executions. In this paper we will consider fractions representing 20, 40, 60, and 80 percent of the total unique references seen during profile runs.

The **profile** and **locality** strategies use execution profiling to determine what instructions are likely to be executed. Note that this is an incomplete methodology for real-time code, but is sufficient for best-effort code in which we seek to improve the expected case. As will be seen, we can also use profile information to intelligently layout code to reduce conflict misses.

3.2. Implementation

We now consider the implementation details of a segmented cache. Figure 2 shows the organization of a typical direct-mapped cache augmented with a unit, surrounded by a dashed rectangle, that performs segmentation. Certain bits are extracted from the address and used to index into the tag and data arrays. Tags are compared to make sure the data entry corresponds the desired address rather than some other address that maps to the same set. As can be seen, only a transformation in index bits is required to implement segmentation.

Segment sizes and offsets can be stored in each thread's status registers. At context switch time, the segment size and offset of the incoming thread can be loaded into the cache mapping unit.

As mentioned earlier, a cache of any kind incurs both access time and size overheads compared to a RAM. In our case, the access time overhead is due to both the index bit transformation and tag comparison. Whether this increased access time is on the critical path and reduces clock rate is a design-specific consideration. If it were (and we would expect it to be on a simple data processor), an extra processor pipeline stage could be added in order to keep from hurting the clock rate (the IXP1200 data processor, for example, uses a 5-stage pipeline and this scheme might require a sixth stage). The size overhead is due to the tag array; each entry requires some number of tag bits not needed in a RAM. For the example shown in Figure 2, each 64-bit data entry requires an 18-bit tag entry, which results in overhead of approximately 28%.

Whether these overheads are worthwhile are, again, design-specific decisions depending on a wide range of factors. Given the dominance of caches in general-purpose and embedded systems, we expect that future NP data processors will pay these overheads in order to gain the benefits of unrestricted program size.

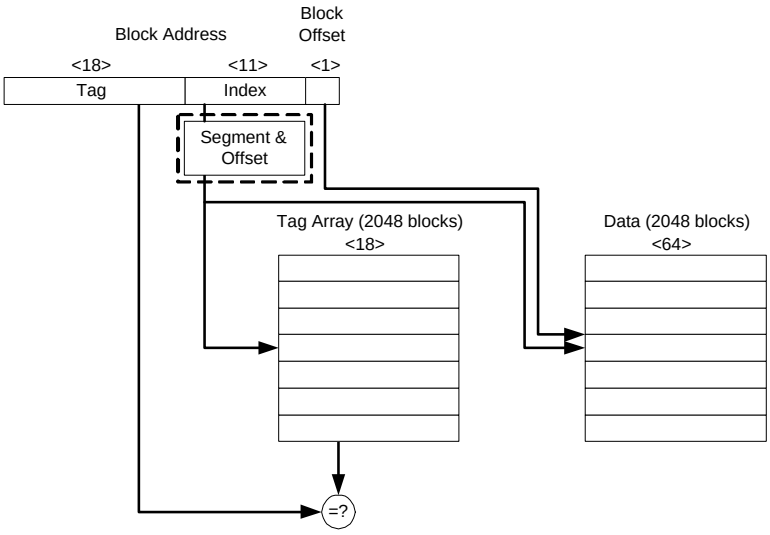

Figure 2. A sample cache organization with 32-bit addresses, 2K entries, 32-bit blocks, and two blocks per set. The segmented cache implementation only requires a transformation on the index bits.

3.3. Address Mapping

If all segment sizes were a power of 2, then the mapping function would be trivial: extract the index bits and use those to index into the tag and data arrays (as is done with normal caches, all of which are sized to be a power of 2); this is a highly efficient implementation of the modulo operator.

For arbitrary segment sizes, however, another approach must be taken since the modulo operation requires an integer division, rather than a simple bit selection, when the modulus is not a power of 2. To illustrate the problem, first consider a segment with 8 entries. The mod operation can be implemented by simply using the last (i.e., least signficant) 3 bits of an address as an index to select one of the $2^3 = 8$ sets. Now consider a segment with 7 entries. 3 bits are still needed to encode the 7 entries, but one of the 3 bit combinations will not correspond to a valid entry (i.e., there are only 7 entries but 8 possible 3-bit sequences). In fact, the invalid entry will map to an entry in another segment, a situation we certainly want to avoid.

One solution would be to implement digital division and calculate the remainder directly. For our purposes, however, a simpler approach is preferable. The following pseudo-code implements the calculation, where N is the segment size and the selected-bits are the bits taken from the address which encode the address (the number of bits needed is equal to the number of bits needed to encode the segment size, i.e., $\log_2 N$, and can be different for each thread).

```
if selected-bits >= N
   set = selected-bits - N + offset
else
   set = selected-bits + offset
```

The idea is to detect an invalid bit sequence (i.e., one that is greater than the segment size) and generate a valid bit sequence from it. The computation requires adders and a comparator, but is parallel and should permit fast implementation.

3.4. Enforcing Instruction Memory Bandwidth Limits

Since the path to memory where the rest of the program is stored (i.e. from which misses are serviced) is likely to be a resource shared by real-time program data or I/O accesses, it might be necessary to limit interference from instruction cache misses. There are two considerations. The first is to keep the required instruction memory bandwidth at a level sufficient to meet any real-time constraints. The second task is to keep the required instruction memory bandwidth deterministic so that adequate provisioning of system resources can be performed.

To both of these ends, one could employ an instruction fetch throttle that determines the bandwidth allocated to instruction fetches. In this way, the instruction memory request rate can be set by the system and thereby bounded for the purpose of system provisioning. One policy, for example, would be to give in-

struction access priority over all other memory transactions. The actual amount of instruction memory bandwidth allocated will be a system-level question, while the bandwidth required will be a consequence of program size, execution pattern, thread scheduling pattern, and region size. We plan to explore mechanisms to support such policies in future work.

4. Experimental Evaluation

In the following sections, we evaluate the segmented instruction cache and demonstrate effective sizing strategies and miss reduction techniques.

4.1. Benchmark Programs and Methodology

Our performance simulations are trace-driven. We use several programs which have been made publicly available by researchers [1, 2]; these are described in Table 1. Most of the programs are typical of networking codes: searching, sorting or extracting values, data validation, and encryption. The rest are numeric in nature (e.g., FFT and ludcmp). All are C-based programs and were compiled on a Sun Solaris 8 machine with GCC version 2.95.1. The instruction traces were gathered by running Gnu gdb version 4.15.1.

For the experimental cache results reported in this paper, we used our own cache simulator which has been validated against the DineroIV [6] cache simulator. The simulator is implemented in Python [16] and is easily extensible and can flexibly model a wide variety of memory system features and organizations. A custom simulator was necessary since our cache explorations involve novel (e.g., non-uniform cache mapping) and atypical (e.g., cache set counts that are not a power of two) features. In all experiments, a line size of 1 word (32 bits, i.e., one instruction) is used.

4.2. Segment Sizing

In this section, we investigate the effectiveness of segment sizing strategies. Recall, that the **program** sizing strategy will yield no cache misses at all, as will **profile** provided that the profiling inputs are correct; the former will be used from programs that cannot tolerate misses, and the latter can be used for programs that should not miss but can afford to do so occasionally. The **locality** strategy will, by definition, incur some misses. It is the most optimistic and anticipates a high amount of locality in the reference stream. Thus, in our cache experiments, only the locality strategy will be examined. Table 2 reports program-specific segment sizes for each of these strategies.

While gathering good profiles is an important task in practical software engineering, it is outside of our interest in this study. We use the profile inputs provided with each benchmark for our evaluations. Note that our emphasis is not in measuring the accuracy of the profiles, but rather in determining the performance of the segmented instruction cache given good profile information.

To gain initial intuition about how much locality is present, we first consider the distribution of instruction references in our profile runs. Figure 3 depicts the distribution of dynamic instruction references over static instructions. For example, the bottom-most stacked segment in the binsearch column indicates that 10% of the unique static instructions seen during execution account for 20% of the dynamic references. The most extreme example is tstdemo, in which 95% of dynamic instructions are caused by fewer than 5% of the static instructions. On average, however, it appears that around 60% of the static instructions account for 95% or more of the dynamic references. This suggests that if we can cache those 60% and keep them from conflicting with one another, the remaining 40% cannot impose too many misses if they are not kept in the cache.

Table 3 reports miss rates for the **locality** segment sizing strategy, both before and after pre-loading, and categorizes the misses for the pre-loaded miss rate. There are several points to be made.

- Miss rate improves with increasing segment sizes.
- Miss rate improves, as expected, with pre-loading.
- Miss rates (even with pre-loading) are quite high (e.g., 14% or higher with locality 60) for five of the programs: binsearch, FFT, qsort, qurt, and select. Around 25% of misses for those programs are due to conflicts.

Recall that of the three types of uniprocessor cache misses, conflicts are the only type we can manipulate further for a given segment size (some compulsory misses are avoided via pre-loading, and capacity misses are determined by segment size). Since they can be manipulated, we discuss the sources of conflicts and techniques for reducing them in the following sections.

4.3. Sources of Conflict Misses

Cache conflict misses can result when two or more addresses map to the same set; this can be called a *spatial* conflict. Spatial conflicts are a necessary but insufficient condition for conflict misses: there must

Table 1. Sample programs discussed in this study.

Program	Static I. C.	Dyn. I. C.	Unique Dyn. I.C.	Description
binsearch	199	175	79	Binary search over 15 integers.
chk_data	99	295	66	Example from Park's [14] thesis that finds the first non-zero entry in an array.
CRC	442	71633	258	Cyclic redundancy check.
DES	1058	110217	929	Data encryption standard [12].
FFT	835	3915	305	Fast Fourier transform.
fibcall	168	557	48	Sum the first 60 Fibonacci numbers.
isort	209	2215	89	Insertion sort of 10 integers.
ludcmp	761	8851	631	LU decomposition of linear equations.
matmul	274	8707	158	Matrix multiplication.
qsort	574	2649	426	Quicksort.
qurt	492	1714	365	Finding roots of quadratic equations.
select	520	2959	367	Select k largest integers from a list.
tstdemo	2558	1.67M	2010	Using ternary search trees [1] to find all words in a 20K-word dictionary that are within a Hamming distance 3 of "elephant".

Table 2. Sample segment sizes.

Program	**program**	**profile**	**locality**				
			20	40	60	80	95
binsearch	199	79	16	32	48	64	76
chk_data	99	66	14	27	40	53	63
CRC	442	258	52	104	155	207	246
DES	1058	929	186	372	558	744	883
FFT	835	305	61	122	183	244	290
fibcall	168	48	10	20	29	39	46
isort	209	89	18	36	54	72	85
ludcmp	761	631	127	253	379	505	600
matmul	274	158	32	64	95	127	151
qsort	574	426	86	171	256	341	405
qurt	492	365	73	146	219	292	347
select	520	367	74	147	221	294	349
tstdemo	2558	2010	402	804	1206	1608	1910

also be a *temporal* conflict between the addresses (i.e., the references must be interleaved with one another over time). If the addresses are not referenced near to one another in time, then few, if any, conflict misses will result.

In a multithreaded processor, there are two possible sources of conflict misses: intra-thread and inter-thread. Intra-thread conflicts arise when addresses within one thread conflict. Inter-thread conflicts are possible when different threads share a segment and their address request patterns conflict; we consider segment sharing like this in Section 4.6

If we know in advance which paths are most likely to be executed, then it is possible for us to schedule our program into memory in such a way as to minimize spatial conflicts between instructions on the frequently executed paths; we consider this topic in the following section. There are also well-known hardware techniques for reducing conflict misses as well, and we consider these in Section 4.5.

4.4. Profile-Driven Code Scheduling to Reduce Misses

As can be seen in Figure 3, a fraction of static instructions are responsible for a majority of the dynamic references. By using profile information, we can identify those instructions that occur most frequently and position them in memory, so that they conflict with other infrequently executed (or ideally nonexistent) instructions. This general technique is called code-scheduling or code-reordering and is well studied[15].

Code scheduling is a particularly appropriate technique for NP programs since so much is known and statically fixed at compile time. Using this approach, we can schedule the code in our benchmark programs. The resulting effect on miss rate is shown in Table 4. There are, again, several points to be made.

Table 3. Cache results for sample segment sizes.

Program	**locality**	Miss Rate	With Pre-Loading			
			Miss Rate	Compulsory	Capacity	Conflict
binsearch	20	0.98	0.89	0.40	0.60	0.00
	40	0.84	0.66	0.41	0.59	0.00
	60	0.50	0.23	0.78	0.00	0.23
	80	0.45	0.09	1.00	0.00	0.00
chk_data	20	0.94	0.89	0.20	0.80	0.00
	40	0.42	0.33	0.40	0.00	0.60
	60	0.22	0.09	1.00	0.00	0.00
	80	0.22	0.04	1.00	0.00	0.00
CRC	20	0.39	0.39	0.01	0.93	0.06
	40	0.17	0.17	0.01	0.00	0.98
	60	0.00	0.00	0.69	0.11	0.19
	80	0.00	0.00	0.76	0.12	0.12
DES	20	0.16	0.16	0.04	0.34	0.62
	40	0.03	0.03	0.20	0.01	0.78
	60	0.01	0.00	0.89	0.00	0.11
	80	0.01	0.00	0.84	0.00	0.16
FFT	20	0.74	0.73	0.09	0.91	0.00
	40	0.64	0.61	0.08	0.40	0.52
	60	0.19	0.14	0.22	0.50	0.28
	80	0.10	0.03	0.46	0.00	0.54
fibcall	20	0.90	0.88	0.08	0.92	0.00
	40	0.09	0.05	1.00	0.00	0.00
	60	0.09	0.03	1.00	0.00	0.00
	80	0.09	0.02	1.00	0.00	0.00
isort	20	0.93	0.93	0.03	0.97	0.00
	40	0.48	0.47	0.05	0.91	0.04
	60	0.07	0.05	0.34	0.66	0.00
	80	0.04	0.01	1.00	0.00	0.00
ludcmp	20	0.19	0.17	0.33	0.67	0.00
	40	0.09	0.06	0.74	0.24	0.02
	60	0.09	0.05	0.60	0.00	0.40
	80	0.07	0.01	1.00	0.00	0.00
matmul	20	0.88	0.88	0.02	0.98	0.00
	40	0.19	0.18	0.06	0.41	0.52
	60	0.13	0.12	0.06	0.00	0.94
	80	0.13	0.12	0.03	0.00	0.97
qsort	20	0.53	0.50	0.26	0.73	0.01
	40	0.48	0.42	0.23	0.76	0.01
	60	0.42	0.33	0.20	0.80	0.00
	80	0.30	0.17	0.18	0.37	0.44
qurt	20	0.60	0.56	0.31	0.30	0.40
	40	0.33	0.25	0.52	0.48	0.00
	60	0.31	0.18	0.48	0.35	0.18
	80	0.23	0.06	0.68	0.02	0.30
select	20	0.89	0.86	0.12	0.88	0.00
	40	0.65	0.60	0.12	0.88	0.00
	60	0.30	0.23	0.21	0.12	0.67
	80	0.20	0.10	0.25	0.00	0.75
tstdemo	20	0.03	0.03	0.03	0.05	0.91
	40	0.03	0.03	0.03	0.01	0.97
	60	0.00	0.00	0.30	0.03	0.67
	80	0.00	0.00	0.25	0.00	0.75

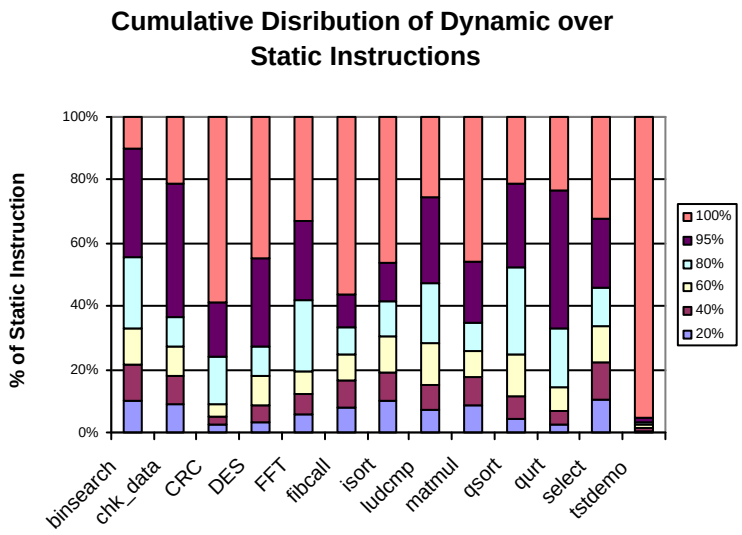

Figure 3. This graph reports the percentage of static instructions seen during execution that are responsible for 20%, 40%, 60%, 80%, and 95% of the dynamic instructions observed in a profile run.

- Miss rates always decrease as segment sizes increase.
- All **locality** 80 pre-loaded miss rates are at or below 10%.
- Conflict misses rarely dominate the other types, and only do so at very low miss rates. It appears that most conflicts have been avoided.

Figure 4 directly compares the miss rates for the **locality** 60 segment sizing strategy, the only strategy we evaluate in the remainder of the paper. All miss rates for that strategy are now below 26% with most being below 10%.

Nearly all programs benefit from scheduling, with miss rates reduced by between 10% and 60%. Two numeric programs, however, DES and ludcmp, have increases of 3% and 5%, respectively. Both of these programs experience increased conflict misses at this segment size when code is scheduled according to profiled execution frequency.

4.5. Using Set-Associativity to Reduce Misses

Cache set-associativity is a hardware-based approach for reducing conflicts. In a set associative cache, each address maps to one set that contains m entries (in an m-way cache); these m entries are accessed in a fully associative manner (i.e., the requested item can be found in any of the positions and they must all be searched in parallel). So long as no more than m addresses conflict in each set, conflict misses can be avoided.

Associativity is a feature orthogonal to segmentation; in a segmented cache, it doesn't matter whether a set is a single item or a collection of them. In our next set of experiments, we measure the effect of associativity on conflict misses both before and after code scheduling.

The results are shown in Figure 5. In the figure, conflict misses for each cache organization are reported as normalized to the number of conflicts seen in a direct-mapped cache of the same capacity.

We make the following observations.

- Set-associativity alone is ineffective at reducing conflict misses. The first part of Figure 5 shows that conflicts actually increase at least as often as they decrease. This counter-intuitive result is due to the fact that when associativity increases, so do the number of potentially conflicting addresses that map to a single set; if an increase in associativity brings into the set a more frequent conflicting address, misses can increase. Of course, this is only likely to happen in *full* caches (e.g., caches that utilize their full capacity), as these caches are by design.
- Set-associativity is a clear benefit when used with

Table 4. Cache results after code scheduling.

Program	**locality**	Miss Rate	With Pre-Loading			
			Miss Rate	Compulsory	Capacity	Conflict
binsearch	20	0.99	0.90	0.40	0.60	0.00
	40	0.57	0.38	0.70	0.30	0.00
	60	0.48	0.21	0.86	0.00	0.14
	80	0.47	0.10	0.83	0.00	0.17
chk_data	20	0.91	0.86	0.20	0.80	0.00
	40	0.22	0.13	1.00	0.00	0.00
	60	0.22	0.09	1.00	0.00	0.00
	80	0.23	0.05	0.93	0.00	0.07
CRC	20	0.37	0.37	0.01	0.99	0.00
	40	0.00	0.00	0.80	0.19	0.01
	60	0.00	0.00	0.87	0.13	0.00
	80	0.00	0.00	1.00	0.00	0.00
DES	20	0.16	0.16	0.04	0.37	0.59
	40	0.05	0.05	0.10	0.01	0.89
	60	0.04	0.03	0.10	0.00	0.90
	80	0.01	0.01	0.31	0.00	0.69
FFT	20	0.56	0.55	0.11	0.88	0.00
	40	0.34	0.31	0.15	0.82	0.03
	60	0.14	0.09	0.34	0.25	0.40
	80	0.09	0.03	0.56	0.00	0.44
fibcall	20	0.90	0.88	0.08	0.92	0.00
	40	0.09	0.05	1.00	0.00	0.00
	60	0.09	0.03	1.00	0.00	0.00
	80	0.09	0.02	1.00	0.00	0.00
isort	20	0.98	0.97	0.03	0.97	0.00
	40	0.42	0.40	0.06	0.88	0.06
	60	0.05	0.03	0.52	0.24	0.24
	80	0.04	0.01	1.00	0.00	0.00
ludcmp	20	0.32	0.30	0.19	0.35	0.46
	40	0.14	0.11	0.39	0.41	0.20
	60	0.14	0.10	0.29	0.00	0.71
	80	0.07	0.02	0.80	0.00	0.20
matmul	20	0.90	0.89	0.02	0.97	0.02
	40	0.11	0.10	0.11	0.82	0.07
	60	0.03	0.02	0.33	0.02	0.66
	80	0.02	0.00	0.94	0.00	0.06
qsort	20	0.60	0.57	0.23	0.65	0.13
	40	0.51	0.44	0.22	0.73	0.05
	60	0.35	0.26	0.25	0.68	0.07
	80	0.21	0.08	0.42	0.31	0.28
qurt	20	0.38	0.34	0.50	0.50	0.00
	40	0.31	0.22	0.57	0.41	0.02
	60	0.27	0.14	0.61	0.25	0.15
	80	0.24	0.07	0.61	0.23	0.16
select	20	0.91	0.88	0.11	0.89	0.00
	40	0.45	0.40	0.19	0.81	0.00
	60	0.17	0.09	0.54	0.34	0.12
	80	0.13	0.03	0.78	0.00	0.22
tstdemo	20	0.00	0.00	0.42	0.48	0.10
	40	0.00	0.00	0.68	0.18	0.14
	60	0.00	0.00	0.72	0.07	0.21
	80	0.00	0.00	0.70	0.00	0.30

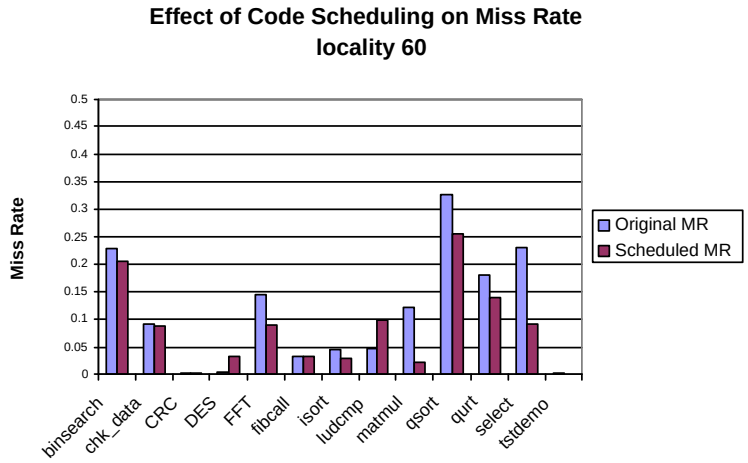

Figure 4. Compares miss rates before and after code scheduling with the locality 60 segment sizing strategy.

code scheduling, more than half the time halving the number of conflict misses.

4.6. Segment Sharing

As discussed in Section 4.3, it is possible, and indeed common, for multiple thread contexts to execute the same program. In this case, it makes sense to assign each context to the same cache segment. While economizing on instruction cache space, this situation raises the possibility of inter-thread conflicts. In this section, we measure the effect of this type of sharing on miss rate. (Another possibility includes mapping different programs into the same segment for programs to share, but we leave this to future work.)

Since segments are shared, our simulation methodology must change somewhat. If T threads are sharing a segment, then each of those threads are likely to be at different points of program execution. In our experiments, we begin simulation by assigning each thread a random start location in the instruction trace. Each thread then executes, one at a time in round robin fashion, until a predetermined number of instructions have been fetched. In these experiments, we limit each thread to $1/T$th of the total number of instructions in the original trace (so that a segment sharing profile run fetches as many instructions as a standalone profile run). In addition to a round-robin scheduling policy (which is the one implemented in the Intel IXP NPs), each thread executes a random number of instructions (sampled from a set of run-lengths normally distributed around 7 instructions) before swapping out.

Figure 6 reports miss rates when segments are shared between multiple threads. Results are shown direct-mapped and 4-way caches sharing 2, 4 and 8 threads.

We make the following observations.

- Miss rates can increase significantly, in the most extreme cases of FFT and ludcmp, nearly doubling.
- Set-associativity is always an improvement, but sometimes only negligibly so (e.g., binsearch 2 and 4 threads).
- In a few cases (e.g., DES, qsort, qurt), miss rate improves due to constructive interference.

Clearly, the decision to instantiate multiple threads to execute a given program only makes sense if it leads to improved performance. If profiling information were to suggest (as it does for several cases in Figure 6) that additional threads might decrease performance, then, indeed, fewer threads should be instantiated.

The problem of segment sharing on a segmented cache is basically equivalent to generic cache sharing on multithreaded processors [9], an area of research where some code scheduling work has been done. However, little of that work is applicable here since it is unclear whether it applies to small, full caches such as these; we plan to explore this issue further in future work. For example, it may be that only caching frequently executed instructions (based on profile information) would greatly reduce inter-thread conflicts.

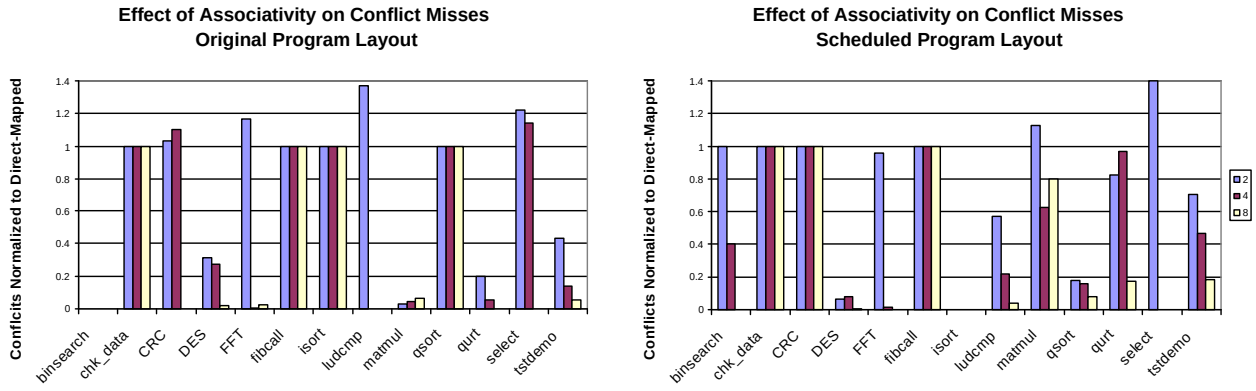

Figure 5. Reports the effect of set-associativity on conflict misses both before (left) and after (right) code scheduling with the locality 60 segment sizing strategy. Results are normalized to the number of conflict misses seen in a direct-mapped cache; 2-, 4-, and 8-way set-associativities are reported.

5. Related Work

There is a considerable body of work in the independent areas of real-time cache analysis and multithreaded/multiprogrammed cache analysis and design.

In the real-time literature, several groups have studied worst-case execution time (WCET) cache analysis [11, 13, 5]. Each of these techniques propose different analytical methods for bounding the performance of a given program on a given processor and cache organization. The methods are descriptive and do not prescribe improvements in program structure should the WCET be too great.

Instruction delivery in multithreaded processors were first considered in the design of the first multithreaded computers [18] and multicomputers [17]. More recent studies evaluate workstation and server cache performance and requirements [3] on multithreaded architectures, as well as the previously mentioned work on compilation for instruction cache performance on multithreaded processors [9].

Column caching [4] is a dynamic cache partitioning technique in which individual associative ways are allocated for various purposes, such as to threads or programs for guaranteed performance. The drawback is that high associativities are need for a high degree of allocation. In any case, segmentation, which does not require set associativity, is orthogonal to column caching and could be used to increase the granularity of allocation.

6. Conclusions and Future Work

In this paper, we proposed the use of segmented instruction caches along with profile-driven code scheduling to provide flexible instruction delivery to real-time and non-real-time threads running on the same multithreaded procesor. This technique is particularly useful in NP data processors, which are multithreaded yet inhibited by a fixed size control store. The segmented instruction cache allows real-time programs to map into a private segment large enough to avoid misses while allowing non-real-time programs to suffer misses while keeping all cache conflicts limited to within individual segments. This removes program size restrictions on non-real-time code without sacrificing guaranteed instruction delivery to real-time programs. Several program-specific segment sizing strategies were evaluated, and code scheduling was seen to be an effective method for removing a majority of conflict misses and often reducing miss rates on a selection of programs by a range of 10% to 60%.

We plan to consider a number of additional topics in future work. These include: avoiding index calculations and tag checks (completely for real-time segments, and via speculation otherwise), mechnisms to shape instruction fetch bandwidth, improving shared segment miss rates, measuring sensitivity to cache parameters (e.g., block size), investigating the benefits of dynamically changing segment sizes at run-time (both for these statically-composed threads as well as dynamically composed scenarios), providing sharing between segments (e.g., for shared code libraries), and exploring the use of segmentation in data caches.

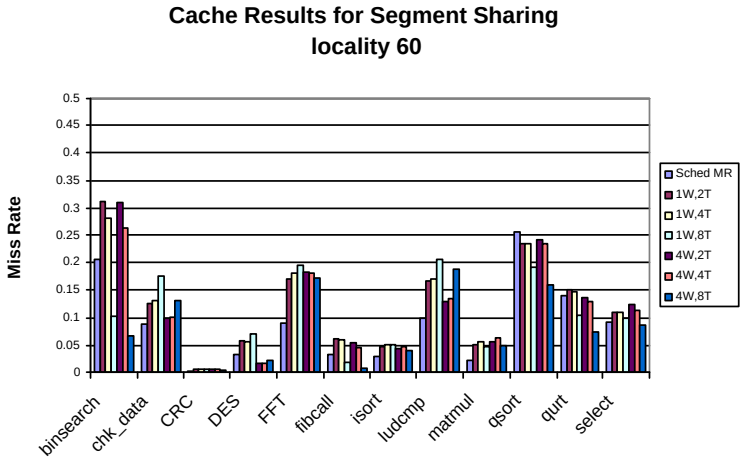

Figure 6. Reports miss rates for direct-mapped and 4-way set associative segments when 2, 4 and 8 threads share a given segment when code is scheduled with the locality 60 segment sizing strategy. Among the legend entries, *1W* indicates a direct-mapped segment, and *2T* indicates two active threads.

References

[1] Bentley and Sedgewick. Fast algorithms for sorting and searching strings. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1997.

[2] C-LAB, Siemens AG, and Universitat Paderborn. C-LAB WCET Benchmarks. `http://www.c-lab.de` , 2003.

[3] Y. Chen, M. Winslett, S. Kuo, Y. Cho, M. Subramaniam, and K. E. Seamons. Performance modeling for the Panda array I/O library. In *Proceedings of Supercomputing '96*. ACM Press and IEEE Computer Society Press, 1996.

[4] D. Chiou, P. Jain, S. Devadas, and L. Rudolph. Dynamic cache partitioning via columnization. In *Proceedings of Design Automation Conference, Los Angeles*, June 2000.

[5] J. Engblom and A. Ermedahl. Modeling complex flows for worst-case execution time analysis. In *Proc. of 21st IEEE Real-Time Systems Symposium (RTSS'00)*, 2000.

[6] M. Hill and J. Elder. DineroIV Trace Driven Uniprocessor Cache Simulator. `http://www.cs.wisc.edu/markhill/DineroIV`, 2003.

[7] M. D. Hill. Aspects of cache memory and instruction buffer performance. Ph.D. Dissertation, Tech. Report UCB/CSD 87/381, Computer Sciences Division, UC-Berkeley, Nov. 1987.

[8] Intel Corp. Intel IXP Family of Network Processors. `http://developer.intel.com`, 2001.

[9] R. Kumar and D. M. Tullsen. Compiling for instruction cache performance on a multithreaded architecture. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 419–429. IEEE Computer Society Press, 2002.

[10] V. Kumar, T. Lakshman, and D. Stiliadis. Beyond best effort: Router architectures for the differentiated services of tomorrow's Internet. *IEEE Communications Magazine*, pages 152–164, May 1998.

[11] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *Proceedings of the IEEE Real-Time Systems Symposium*, 1996.

[12] National Bureau of Standards. Data Encryption Standards. FIPS Publication 46, U.S. Dept. of Commerce, 1977.

[13] G. Ottosson and M. Sjödin. Worst-case execution time analysis for modern hardware architectures. In *ACM SIGPLAN 1997 Workshop on Languages, Compilers, and Tools for Real-Time Systems (LCT-RTS'97)*, 1997.

[14] C. Y. Park. *Predicting Deterministic Execution Times of Real-Time Programs.* PhD thesis, University of Washington, August 1992.

[15] K. Pettis and R. C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation (SIGPLAN '90)*, pages 16–27, June 1990.

[16] Python. The Python programming language. on the web, 2002. `http://www.python.org`.

[17] B. Smith. Architecture and applications of the hep multiprocessor computer system. In *4th Symp. on Real Time Signal Processing*, pages 241–248, August 1981.

[18] J. Thornton. *Design of a Computer: The Control Data 6600.* Scott, Foresman and Co., 1970.