Des Autom Embed Syst (2010) 14: 309–326
DOI 10.1007/s10617-010-9058-y

Reducing impact of cache miss stalls in embedded systems by extracting guaranteed independent instructions

Garo Bournoutian · Alex Orailoglu

Received: 11 May 2010 / Accepted: 24 June 2010 / Published online: 21 July 2010

Abstract Today, embedded processors are expected to be able to run algorithmically complex, memory-intensive applications that were originally designed and coded for general-purpose processors. As such, the impact of memory latencies on the execution time increasingly becomes evident. All the while, it is also expected that embedded processors be power-conscientious as well as of minimal area impact, as they are often used in mobile devices such as wireless smartphones and portable MP3 players. As a result, traditional methods for addressing performance and memory latencies, such as multiple issue, out-of-order execution and large, associative caches, are not aptly suited for the mobile embedded domain due to the significant area and power overhead. This paper explores a novel approach to mitigating execution delays caused by memory latencies that would otherwise not be possible in a regular in-order, single-issue embedded processor without large, power-hungry constructs like a Reorder Buffer (ROB). The concept relies on efficiently leveraging both compile-time and run-time information to safely allow non-data-dependent instructions to continue executing in the event of a memory stall. The simulation results show significant improvement in overall execution throughput of approximately 11%, while having a minimal impact on area overhead and power.

Keywords Embedded processors · Data cache · Pipeline stalls · Compiler assisted hardware

1 Introduction

The prevalence of mobile embedded processors in modern computational systems has grown significantly over the last few years. At the current rate, these mobile devices will become increasingly ubiquitous throughout our society, resulting in a broader range of applications that will be expected to run on these devices. Even today, mobile embedded processors are

G. Bournoutian (✉) · A. Orailoglu
CSE Department, University of California, San Diego, 9500 Gilman Dr. #0404, La Jolla, CA 92093-0404, USA
e-mail: garo@cs.ucsd.edu

expected to be able to run algorithmically complex, memory-intensive applications that were originally designed and coded for general-purpose processors. Furthermore, these processors are becoming increasingly complex to respond to this more diverse application base, as can be seen by the current technological landscape of wireless smartphones. Many embedded processors have begun to include features such as multi-level data caches, but the impact from larger memory access times occurring as a result of cache misses becomes increasingly evident [1].

With the constraints embodied by portable embedded processors, one typically is concerned with high performance, power efficiency, better execution determinism, and minimized area. Unfortunately, these characteristics are often adversarial, and focusing on improving one often results in worsening the others. For example, in order to increase performance, one adds a more complex cache hierarchy to exploit data locality, but introduces larger power consumption, more data access time indeterminism, and increased area. However, if an application is highly regular and contains an abundance of both spatial and temporal data locality, then the advantages in performance greatly outweigh the drawbacks. On the other hand, as these applications become more complex and irregular, they are increasingly prone to thrashing. For example, video codecs, which are increasingly being included in portable wireless devices like smartphones and personal media players, utilize large data buffers and significantly suffer from cache thrashing [2].

In particular, in mobile embedded systems, where power and area efficiency are paramount, smaller, less-associative caches are chosen. Earlier researchers realized that these caches are more predisposed to thrashing, and proposed solutions such as the victim cache [3] or dynamically associative caches [4] to improve cache hit rates. Yet, even with these various solutions to reduce cache misses, today's more aggressive and highly irregular applications inevitably will still suffer from some cache misses, slowing overall application performance.

Furthermore, mobile embedded processors typically do not employ multiple-issue or out-of-order execution, as the hardware overhead is prohibitive. Because of this in-order behavior, when an instruction stalls the pipeline, subsequent instructions are not permitted to execute, even if those instructions are independent and ready for execution. A cache miss is an example of such a stalling instruction, and the processor must wait until the memory reference is resolved before allowing the pipeline to begin issuing and executing instructions again. Each of these memory stalls can result in hundreds of processor cycles wasted in the event that the cache miss needs to read from main memory [5]. As shown in Fig. 1, memory-based instructions account for approximately 35% of the total number of operations executed on average in applications. Thus, minimizing the impact of cache miss stalls can have a significant improvement on overall processor performance and execution time.

In this paper, we propose a solution that will utilize both the compiler and a nominal amount of specialized hardware to extract and allow guaranteed independent instructions to continue execution during memory stalls, which in turn reduces the overall execution time. In this manner, the delay induced by memory stalls can be effectively hidden, allowing the processor to continue safely executing independent instructions similar to an out-of-order processor. Fundamentally, this approach extends the communication link between the compiler and the processor architecture by transferring a small amount of application information directly to the microarchitecture without modifying the existent instruction set. Thus, by combining global application information known at compile-time with run-time cache behavior, we are able to intelligently keep feeding the pipeline with instructions we know are guaranteed to be independent of the memory operation, and thus can be safely executed without any data hazards. We show the implementation of this architecture and provide experimental data taken over a general sample of complex, real-world applications to show

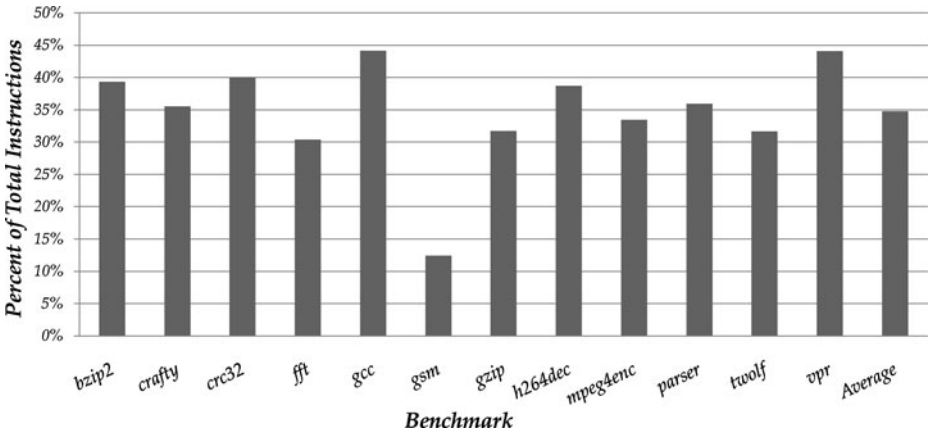

Fig. 1 Average distribution of memory instructions

the benefits of such an approach. The simulation results show significant improvement in execution throughput of approximately 11% by hiding approximately 64% of cache miss stalls, while having a minimal impact on area overhead and power consumption.

2 Related work

In the last ten years, the industrial embedded processor space has seen enormous expansion. Processors provided by companies such as ARM and Tensilica have become increasingly more powerful and complex, and are used in a wide variety of industrial applications. For example, current cellular smartphone technology often incorporates one or more embedded processors, such as the ARM11, ARM Cortex-A8, or Qualcomm Snapdragon Scorpion processor, along with a number of sophisticated specialized DSP processors, such as Qualcomm's QDSP6. These mobile smartphones are expected to handle a wide variety of purposes, from broadband data communication to high-definition audio/video processing, and even real-time GPS tracking. These target applications are becoming increasingly complex and memory-intensive, and numerous techniques have been proposed to address the memory access challenges involved. Unfortunately, embedded processors are often more highly constrained than general-purpose processors, and many general solutions are precluded by these tougher design constraints.

Various techniques have been proposed and used in the computer architecture community to attack the problem of memory stalls. In general, superscalar, out-of-order processors inherently can mitigate memory stalls by leveraging the reorder buffer (ROB) to dynamically allow multiple instructions to issue whenever resources are available and data dependencies have been met [6, 7]. However, as mentioned in [8], even though in-order execution suffers from a 46% performance gap when compared to out-of-order, an out-of-order architecture is not ideal for embedded systems since it is prohibitively costly in terms of power, complexity, and area.

As an alternative to costly out-of-order processors, [9] proposes having the compiler provide reordering by explicitly placing instructions into per-functional-unit delay queues, which in turn still operate in an in-order fashion. While this is less complex than a full out-of-order processor, it still requires a rather large area overhead for the delay queues and does not allow for fine-grained, dynamic adjustment based on runtime cache hit/miss behavior.

Proactively prefetching data into the cache is another well-established approach to reducing the performance impact of memory stalls. Software prefetching methods [10–13] rely on the compiler to statically insert *prefetch* instructions before that actual load instruction to help mitigate potential memory stalls. Unfortunately, these instructions may cause code bloat and increase register pressure. Furthermore, purely software approaches cannot leverage runtime information relating to cache hit/miss behavior, and thus could result in wasted cycles from unnecessary prefetching. Alternatively, hardware prefetching methods [3, 14–16] utilize access patterns to predict cache misses and inject necessary prefetch logic in the hardware. Unfortunately, as with all prediction schemes, these techniques heavily rely on memory access patterns and react poorly to applications with large or irregular memory accesses. Incorrect prediction can lead to cache pollution and a significant performance penalty.

The authors in [17] propose using a combination of compiler and hardware support to prioritize instructions that are needed to keep the pipeline moving, and when none are available, allows for the buffered low-priority instructions to execute. While this proposed scheme is said to be for an in-order embedded processor, it unfortunately still fundamentally relies upon a reorder buffer (ROB) and run-time register renaming, which are the largest contributors of area and power consumption in an out-of-order system. In fact, as exemplified by [18], the 56-entry instruction issue queue in the HP PA-8000 utilizes 20% of the die area, which is impractical for most mobile embedded systems.

3 Motivation

A typical data-processing algorithm consists of data elements (usually part of an array or matrix) being manipulated within some looping construct. These data elements each effectively map to a predetermined row in the data cache. Unfortunately, different data elements may map to the same row due to the inherent design of caches. In this case, the data elements are said to be in "conflict". This is typically not a large concern if the conflicting data elements are accessed in disjoint algorithmic hot-spots, but if they happen to exist within the same hot-spot, each time one is brought into the cache, the other will be evicted, and this *thrashing* will continue for the entire hot-spot.

Given complex and data-intensive applications, the probability of multiple cache lines being active within a hot-spot, as well as the probability of those cache lines mapping to the same cache set, increases dramatically. As mentioned, much prior work has already gone into minimizing and avoiding cache conflicts and thrashing. Nevertheless, as applications continue to become more complex, the working set of data may exceed the capacity of the cache or lack localized regularity, both of which will degrade cache performance and lead to an increased miss rate. Additionally, since mobile embedded processors are constrained to frugal hardware budgets and compact form factors, they often lack complex hardware to allow out-of-order execution (e.g. reorder buffer, register renaming, and multiple-issue). Thus, when these cache misses do occur, the pipeline must stall and wait until the missed cache line is resolved before continuing execution. Given the current trajectory of application complexity expected to run on these mobile embedded systems, the possibility of such cache misses and resulting memory pipeline stalls will become more prevalent and detrimental to execution speed.

To illustrate this point, Fig. 2 shows an excerpt of assembly code from a GSM full-rate speech transcoder compression algorithm, which could be found on a typical GSM cellular phone. This basic block contains a single load instruction, and based on profiling,

Fig. 2 GSM example basic block

```
01: addiu   $18, $29, 16
02: addiu   $19, $0,  320
03: addiu   $17, $29, 336
04: lui     $20, 4096
05: addiu   $20, $20, 604
# Following load 4.07% miss rate
06: lw      $2,  -32064($28)
07: addiu   $4,  $29, 16
08: jalr    $31, $2
```

it was found that this particular instruction encountered a cache miss 4.07% of the time. If one were to assume that the cache miss penalty was 10 cycles and that all instructions take one cycle, the impact of cache misses would amount to an execution cycle increase of 5.09% for this particular basic block. Since this particular basic block was executed 178,009 times, this would result in an additional 9,061 cycles, which translates to about 0.011 ms of additional execution time overhead (assuming an 800 MHz clock rate). Obviously, this example was looking at just a single basic block; a given application would have many more basic blocks, each of which could suffer a similar fate and manifest into a much larger increase in execution time for the entire application.

The goal of this paper is to allow the compiler to analyze the basic block above, noting that the only instruction dependent on the memory load is the jump-and-link (*jalr*). Thus, the compiler can safely rearrange the code above to place the memory load at the beginning and leave the jump-and-link at the end, effectively positioning six instructions between the memory load and the next dependent instruction. This information is then provided to a specialized hardware structure when the application is loaded for execution. This hardware structure will then be able to detect if a cache miss occurred, and instead of blindly stalling the pipeline like before, it can safely know that six instructions can execute while the memory subsystem is resolving the cache miss. Thus, in this example, the execution cycle increase will become 2.04% instead of 5.09%, effectively hiding about 60% of the cache miss penalty.

4 Implementation

In order to address the issue of cache miss induced pipeline stalls within an in-order embedded processor, we plan to strengthen the interaction between the compiler and the underlying hardware microarchitecture. Inherently, there are certain algorithmic features that can more easily be ascertained during static compile-time analysis, such as examining the global program data flow and understanding data interdependencies. This information would be prohibitively costly and complex to generate on the fly during run-time. On the other hand, there are certain behavioral aspects of the application that can only be ascertained during run-time, such as actual cache hit/miss behavior and when a cache miss is resolved. These events cannot statically be known *a priori*, and thus require real-time system information. Given this trade-off between information available at compile-time versus run-time analysis, it becomes evident that some mixture of the two will be essential. The compiler will help glean crucial information regarding our application that would otherwise be too costly to obtain at run-time, and pass this information to the underlying hardware microarchitecture to help the hardware make intelligent decisions in response to run-time events. A high-level overview of this architecture is shown in Fig. 3.

The proposed solution for cache miss pipeline stalls is composed of two parts: a *compile-time* mechanism, which will analyze and reorder the instructions within basic blocks and a

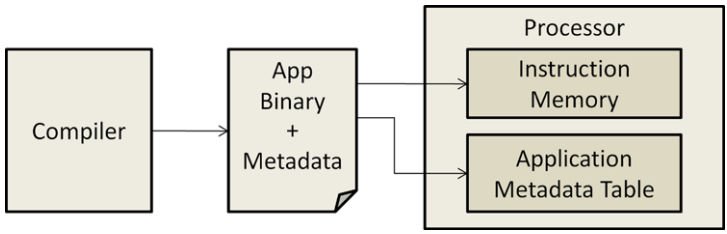

Fig. 3 Overview compiler and hardware interaction

run-time instruction execution mechanism that allows independent instructions to continue running during cache misses. Each part is described in detail in the following sections.

4.1 Compile-time analysis and reordering

The first aspect of our methodology is to have the compiler statically analyze the application and identify those instructions that can safely be executed during a memory stall. Intuitively, these are the instructions that are not data dependent on a memory load and will not cause a hazard if executed prior to the completion of the memory load. Once these instructions are found, they will be aggregated and placed between the load instruction and any subsequent dependent instructions, which will enable us to execute those instructions during a cache miss.

In order to accomplish this analysis, we will leverage the *data flow graph* (DFG) normally used during compilation. The DFG provides an algorithmic representation of the data dependencies between a number of operations. To simplify and bound our analysis, we will operate on a *basic block* granularity, where a basic block is just a segment of code which has only one entry point and one exit point (e.g. no control flow changes exist within the block of code). Each basic block is represented with a DFG $G = (V, E)$, where each $v \in V$ is an instruction in the basic block. There exists a directed edge $e = (v_i, v_j) \in E$ if operation v_i is data dependent on operation v_j within the basic block.

We propose an algorithm that will make use of the DFG to analyze the instruction dependency characteristics and rearrange the instructions in a manner which will place memory loads earlier while pushing instructions dependent on these memory loads later within the basic block. The algorithm to reorder the instructions within a basic block is provided in Fig. 4.

This algorithm must fulfill the following three constraints in order to maintain proper program execution correctness and behavior:

- True data dependencies must be maintained
- The order of store operations must be maintained
- The order of load operations respective to store operations must be maintained

In particular, it is important to note that the constraints listed above with regard to memory store ordering could be relaxed if the effective destination address is statically guaranteed not to conflict with other memory load/store instructions. For simplicity, we conservatively assume that this cannot be guaranteed. However, our algorithm can easily be extended to support this additional compile-time analysis, which would increase the degrees of freedom available for reordering.

After this algorithm completes, the basic blocks will have the following characteristics:

```
ReorderAlgorithm {
  unvisitedSet.initialize( basicBlock.getAllLoadInstrns() );
  visitedStack.clear();
  topPtr = basicBlock.firstInstrnPosition();

  // Optionally remove any anti-dependencies via static register renaming
  // (using a basic register liveness analysis, employing use-define chains)
  // or immediate value adjustment

  while( unvisitedSet.hasElements() ) {
    // Select and remove the load instruction reference which depends on the
    // least number of prior instructions from the unvisitedSet
    currLoadInstrn = unvisitedSet.removeLoadWithLeastNumOfParentInstrnDeps();

    // Bubble the selected load instruction and those instructions it depends
    // on as far towards the topPtr position without violating constraints
    newLoadPosition = BubbleInstrnWithParentsTowardPtr(currLoadInstrn, topPtr);

    topPtr = newLoadPosition + 1;
    visitedStack.push(currLoadInstrn);
  }

  bottomPtr = basicBlock.lastNonControlInstrnPosition();

  while( visitedStack.hasElements() && topPtr != bottomPtr ) {
    currLoadInstrn = visitedStack.pop();
    while( currLoadInstrn.hasDependentInstrns() && topPtr != bottomPtr ) {
      // Select the instruction within the topPtr and bottomPtr which most
      // directly depends on the selected load instruction reference
      myInstrn = SelectInstrnWithinBoundsThatMostDirectlyDependsOnLoadInstrn(
                   currLoadInstrn, topPtr, bottomPtr
                 );

      // Bubble the selected instruction and those instructions that depend on
      // it as far towards the bottomPtr position without violating constraints
      newPosition = BubbleInstrnWithChildrenTowardPtr(myInstrn, bottomPtr);

      bottomPtr = newPosition - 1;
    }
  }
}
```

Fig. 4 Compile-time basic block reordering algorithm

- Memory load instructions as early as possible
- Instructions that depend on memory loads as late as possible
- Instructions that depend on earlier memory loads will occur before instructions that depend on later memory loads
- Memory-independent instructions that can be moved will be placed in between the memory loads and instructions that depend on those loads

To illustrate this point, Fig. 5(a) provides an example of a basic block from the *Fast Fourier Transform* algorithm, along with the corresponding DFG in Fig. 5(b). Running the aforementioned algorithm will reorder the instructions as shown in Fig. 5(c). Observe that instructions `01`, `02`, `09`, and `13` are located between the memory loads and memory-dependent instructions. Furthermore, instruction `04` is located before instruction `07`. The importance of this second observation will be described shortly.

An additional observation relates to instruction `12`. This instruction causes an anti-dependency (write-after-read) on instruction `11` (shown by the dotted line). In the example above, we did not eliminate this dependency. Static register renaming could remove this dependency, but at the cost of an additional instruction and added register pressure. However, an alternative in this instance is to note that instruction `11` [`s.s $f0, 0($5)`] could be

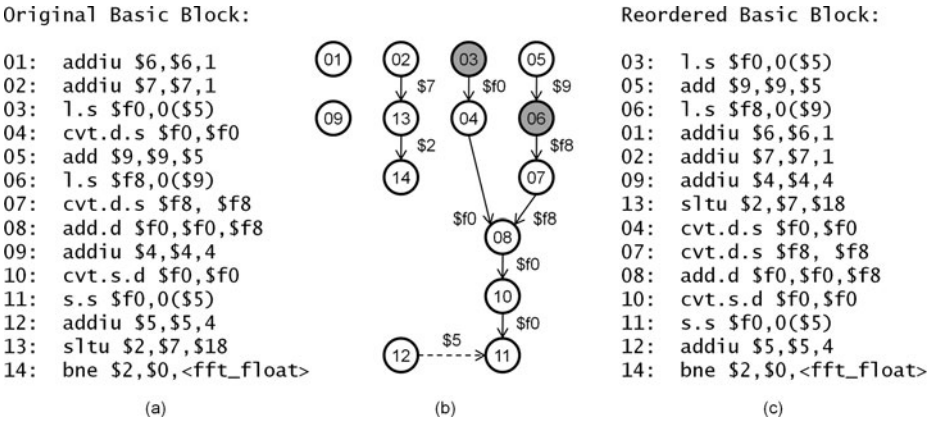

Fig. 5 Excerpt from FFT algorithm with DFG and reordering

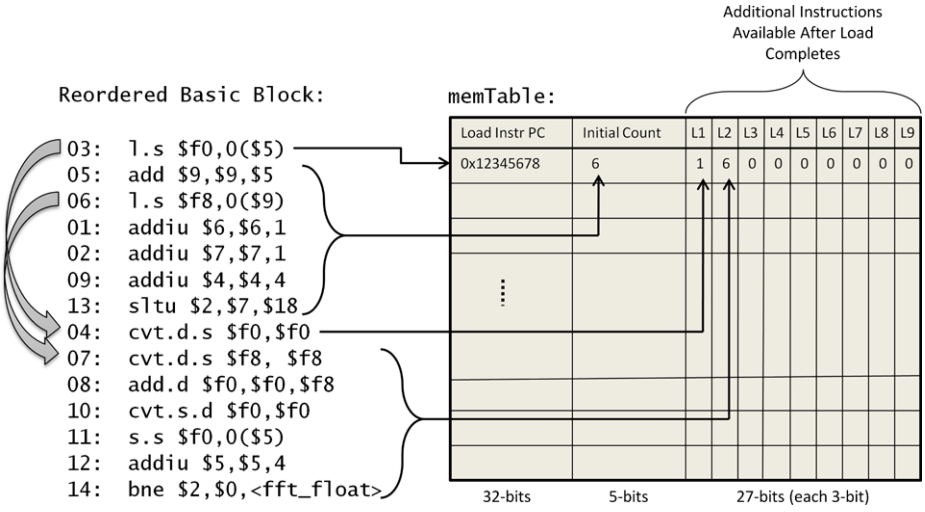

Fig. 6 Basic block memory independence annotation (memTable)

changed to [`s.s $f0, -4($5)`] and then instruction `12` can be moved above it. This more aggressive anti-dependency analysis can help identify additional opportunities that assist in finding instructions that are memory-independent.

Now that the basic block instruction order has been primed, we will selectively annotate those basic blocks which have the greatest potential impact on run-time performance (e.g. the hot-spots). The data structure we will be creating is shown in Fig. 6. Essentially, we need to capture the instruction address (PC) for the first load instruction of the basic block and the number of independent instructions as counted from that initial load instruction. Furthermore, we can capture the increase in independent instructions as we resolve each load instruction in the basic block (up to nine in this example, including the initial load instruction).

The first entry in Fig. 6 corresponds to the instructions shown in Fig. 5(c) as an example of this data structure. As one can see, when the first load instruction (`03`) is encountered,

we know that we can safely execute six additional instructions were a cache miss to occur. Furthermore, once instruction `03` completes, we can execute one more instruction, and when instruction `06` completes, we can execute six more instructions.

This compile-time data structure must then be stored and conveyed to the underlying hardware when the application is executed. To accomplish this, the data structure is encoded into the application binary's text segment at a special reserved address range. When a program binary is loaded onto the processor, this special address range will be read and the corresponding hardware data structure populated with this information. An overview of this concept was shown earlier in Fig. 3. As one can see, the interaction between the compiler and the underlying hardware microarchitecture is augmented to allow this new information regarding memory dependencies to be efficiently conveyed. The hardware mechanism that will detect and enable this independent instruction execution will be described in the next section.

4.2 Run-time stall execution mechanism

Given that the compile-time analysis (described in the previous section) has reordered the instructions and annotated the application's program binary, a hardware mechanism is needed to make use of this information. Intuitively, the goal of this mechanism is to leverage the annotated information provided at compile-time along with real-time information on cache misses and memory stall completion to allow the pipeline to keep executing when it would have otherwise needed to stall. To accomplish this behavior, we will need a nominal amount of additional hardware. It is important to note that the magnitude of this new hardware is greatly smaller than a typical out-of-order processor's reorder buffer (ROB).

The first structure we will need is the memory access table (*memTable*) that will hold the information stored in the application binary. This is essentially what is shown in Fig. 6. Namely, this structure will hold the PC, the initial independent instruction count, and nine additional increments for a given basic block's memory operations. A total entry width of 64-bits is chosen for this purpose. This choice allows for 32-bits for the PC address, 5-bits for the initial instruction count, and nine increment slots, each having 3-bits. This corresponds well to the characteristics of hot-spot basic blocks found in real-world benchmark applications that were analyzed.

In particular, Fig. 7 provides the average number of load instructions present per basic block for our various benchmarks. The average number of load instructions across the aggregate of these benchmarks is approximately 8.75, and hence hardware support for up to nine load instructions was chosen. Similarly, Fig. 8 provides the average number of independent instructions found within basic blocks following their first load instruction, as well as the 90th percentile[1] number of independent instructions. Given that the independent instruction 90th percentile value across the benchmarks is approximately 26.33 instructions, 5-bits (i.e. up to 31 instructions) were alloted for the initial independent instruction count. Additionally, the number of entries in this table can be varied, but based on our analysis of these benchmark applications, 256 entries was chosen to allow us to apply this optimization on up to 256 basic blocks within a given application.

An additional feature of our implementation is that we allow subsequent load instructions to be "executed" during the stall of a previous load. As is typical with most embedded processors, the memory subsystem can only service one memory instruction at a given time.

[1] 90% of the basic blocks analyzed could have all of their independent instructions contained within this value.

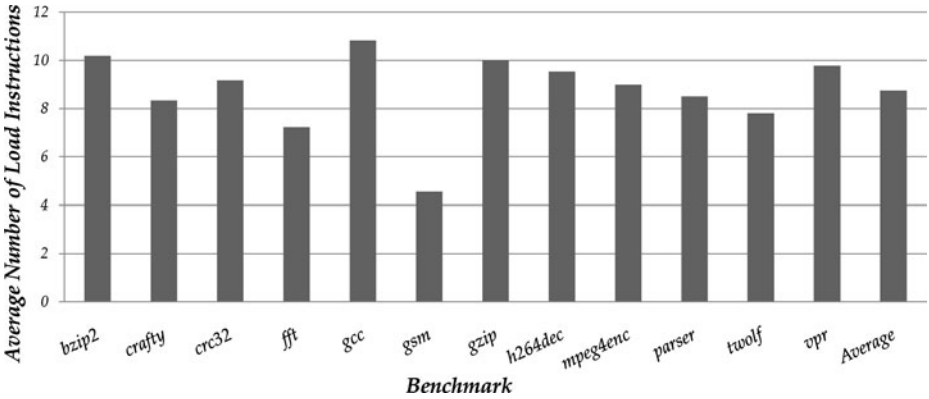

Fig. 7 Average number of load instructions per basic block

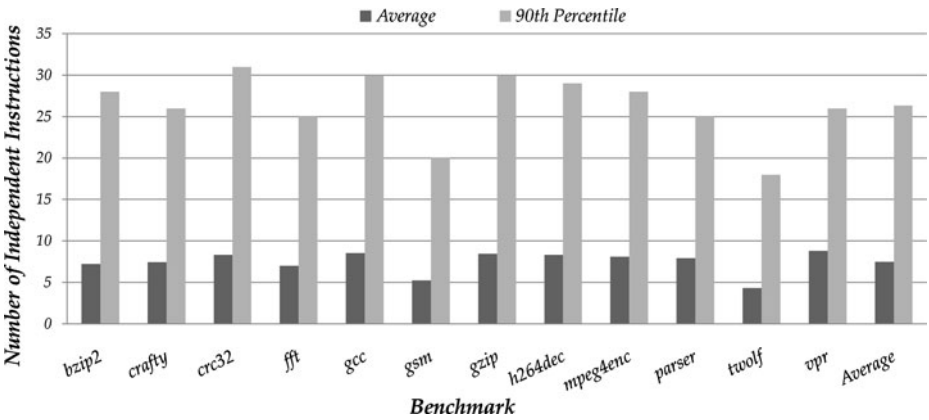

Fig. 8 Average and 90th percentile number of independent instructions per basic block

Thus, it is not possible to execute a load while a prior load is still waiting on a cache miss (e.g. this would be a structural hazard). Nevertheless, instead of always assuming that a load may stall, which requires constraining the number of independent instructions found at compile-time to halt at the first memory instruction, we chose to take an innovative hardware approach, which will be described shortly. With respect to memory stores, we assume the use of a write-back buffer which can be serviced even while the memory subsystem is resolving a memory load, as is common practice in modern embedded processors.

The essential issue is that there *may* be a cache miss on a given load instruction at run-time, but then again, there may not be. Instead of always assuming the worst-case that is necessary at compile-time, we can leverage this run-time behavior to overcome this issue. We propose the addition of a small circular load instruction buffer (*loadBuf*). The purpose of this buffer is to accumulate the effective address, destination register, and memory-specific opcode for a load instruction when the memory subsystem is occupied. For example, if one were to have two back-to-back load instructions, the first instruction could hit or could miss. If it hits, the next instruction can execute normally. On the other hand, if the first instruction misses, we allow the second instruction to get executed in the pipeline and captured in this new load buffer. Immediately after the first instruction completes, the pipeline will inject the

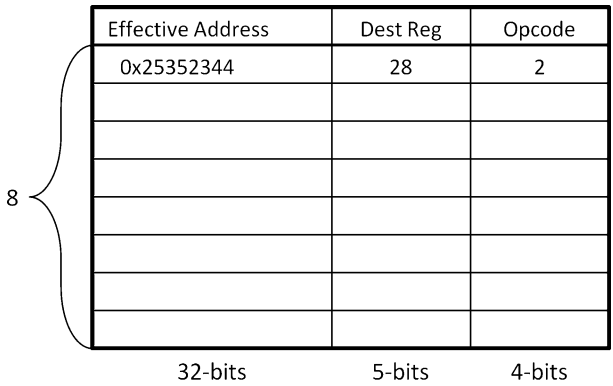

Fig. 9 Load instruction buffer (loadBuf)

buffered memory signals to the memory subsystem and begin the second load. The structure of this load buffer is shown in Fig. 9. We calculated the entry width to be 41-bits, to allow for the typical 32-bit address, 5-bit register number (either integer or floating-point), and 4-bit opcode (which can handle up to 15 types of memory load instructions). We chose eight entries to correspond to the nine additional increment positions in the *memTable* (since one of those positions is for the initial load instruction). Since this buffer is circular and has eight entries, we would also need a 3-bit index register and 3-bits to keep track of how full the buffer is. Thus, this structure accounts for a total storage increase of 334 bits. This structure can be integrated into the memory subsystem to easily capture subsequent load instructions coming into the subsystem.

Lastly, we need to have a counter register (*counterReg*) to keep track of the number of instructions we executed and an increment register (*incrReg*) to store the additional increment values that may occur in the basic block. The *counterReg* lets us know if we have exhausted the number of guaranteed independent instructions and thus must stall if the memory access is not complete. This *counterReg* will be 7-bits, which will allow the initial 5-bit count, plus the possibility of nine additional 3-bit increases without having an overflow. The *incrReg* is a shift register that will shift out the upper 3-bits and add them to the *counterReg* whenever a memory access completes.

The basic idea at run-time is to compare the PC at the memory stage with the entries in the *memTable*. If there is a match, load the corresponding count number and additional increments stored in the *memTable* into the *counterReg* and the *incrReg*, respectively. If the memory access is a miss, continue executing and decrementing the *counterReg* until it becomes zero. At that point, the pipeline will need to be stalled. If the cache miss resolves before the *counterReg* expires, the *incrReg*'s upper 3-bits are shifted out and added to the *counterReg*. This increase to the *counterReg* indicates the number of additional instructions that can now be executed since the previously-stalled memory reference completed. Additional logic exists to account for buffered loads. The full explanation of the run-time behavioral algorithm is provided in Fig. 10.

An important observation is that this implementation will *never* result in worse cycle throughput than the original, non-optimized architecture. If one were to assume 100% cache hit rates, this implementation will behave identically with regard to instruction throughput when compared against the baseline architecture. Only in the event of cache misses, this implementation will allow additional instructions to be executed in otherwise dead cycles during the memory stall. Furthermore, it is necessary that we utilize a run-time control for the algorithm, since having multi-level caches in our architecture can lead to having dynamic miss penalties that cannot be assumed or anticipated at compile-time alone.

```
In memory (M) stage:
  if( PC matches an entry in memTable ) {
    load memTable entry's count into counterReg
    load memTable entry's increment into incrReg
  }

  if( cacheMissSignal ) {
    if( counterReg > 0 ) {
      if( M stage instruction was load ) {
        latch memory instruction information inside subsystem
            (e.g. effective address and dest. reg)
        kill memory instruction (e.g. inject nop as result
            to WB stage)
      }
      if( EX stage instruction was a load ) {
        if( loadBuf is full ) {
          stall pipeline
        } else {
          push instruction to loadBuf and inject nop
        }
      }
    } else {
      stall pipeline (same behavior as default)
    }
  }
  else {
    if( loadInst just completed -- stall or no stall ) {
      if( M stage instr != instruction that just completed ) {
        create a bubble in M stage (e.g. only shift M & WB
            stages in pipeline)
        place memory data and dest reg as input to pipeline regs
      }
      take upper 3-bits of incrReg and add to counterReg
      left shift incrReg by 3-bits
    }
    if( loadBuf not empty ) {
      create a bubble in EX stage
      insert next load instruction as input to M stage
    }
  }

  if( counterReg > 0 ) {
    decrement counterReg
  }
```

Fig. 10 Run-time independent instruction execution behavioral algorithm

5 Experimental results

In order to assess the benefit from this proposed architectural design, we utilized the SimpleScalar toolset [19]. We chose a representative mobile embedded system configuration, having a 256-set, direct-mapped L1 data cache with a 32-byte line size. This processor model utilized a typical in-order simulation engine, with dedicated L1 caches for data and instruction memory, backed by a unified L2 cache.

Twelve representative benchmark programs from the SPEC CPU2000 suite [20], the MediaBench video suite [21], and the MiBench telecomm suite [22] are used. A listing of these benchmarks and their respective descriptions are provided in Table 1.

Figure 11 shows a metric of how well our algorithm was able to identify and allow independent instructions to fill in the delays caused by memory stalls. As one can see, a large amount of the stalled cycles are able to be utilized using our implementation. The average quantity of dead stall cycles that were able to be recouped to run useful instructions was 64.02%, which is rather substantial.

Table 1 Description of benchmarks

Benchmark	Description
`bzip2`	Memory-based compression algorithm
`crafty`	High-performance chess playing game
`crc32`	32-bit CRC framing hash checksum
`fft`	Discrete fast Fourier transform algorithm
`gcc`	GNU C compiler
`gsm`	Telecomm speech transcoder compression algorithm
`gzip`	LZ77 compression algorithm
`h264dec`	H.264 block-oriented video decoder
`mpeg4enc`	MPEG-4 discrete cosine transform video encoder
`parser`	Dictionary-based word processing
`twolf`	CAD place-and-route simulation
`vpr`	FPGA circuit placement and routing

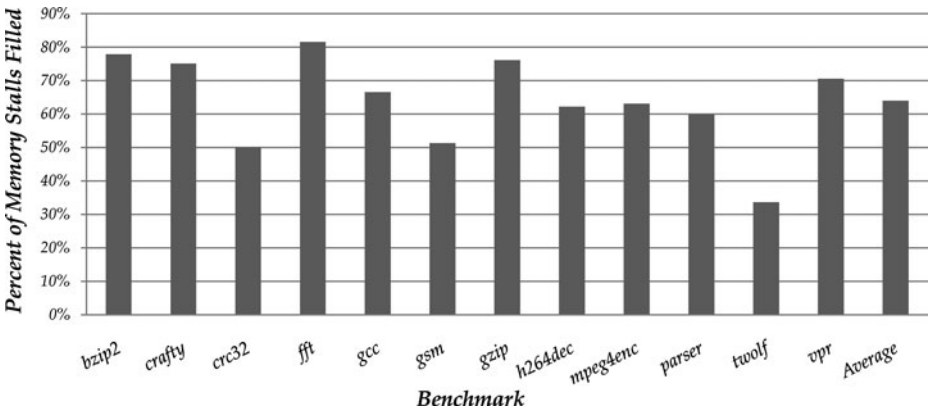

Fig. 11 Success of filling stalls with independent instructions

Given this success in mitigating wasted stall cycles shown above, it is also important to observe what the resulting overall application performance impact would be. To begin with, it is important to look at the baseline cache miss rates. Since the goal of this proposed architecture is to mitigate cache miss induced pipeline stalls, the actual miss rates encountered will directly impact the potential global benefits we hope to achieve. Simply absolving the penalty from a cache miss may not have any tangible improvement on the overall application if there are an infinitesimal amount of cache misses to optimize. Fortunately, there is still a large amount of cache misses that occur within complex, data-intensive applications. Figure 12 provides the L1 cache miss rates observed for the various benchmarks we examined, having an average miss rate of 6.48%. Using this information and the success in filling stalled cycles described earlier, we can now compute the actual overall execution improvement for the application.

Table 2 provides the total number of execution cycles, number of dead execution cycles from cache miss stalls, and percentage of execution overhead due to memory stalls for the baseline architecture. Table 3 provides the total number of execution cycles and number of dead cycles after our architecture allowed independent instructions to execute during mem-

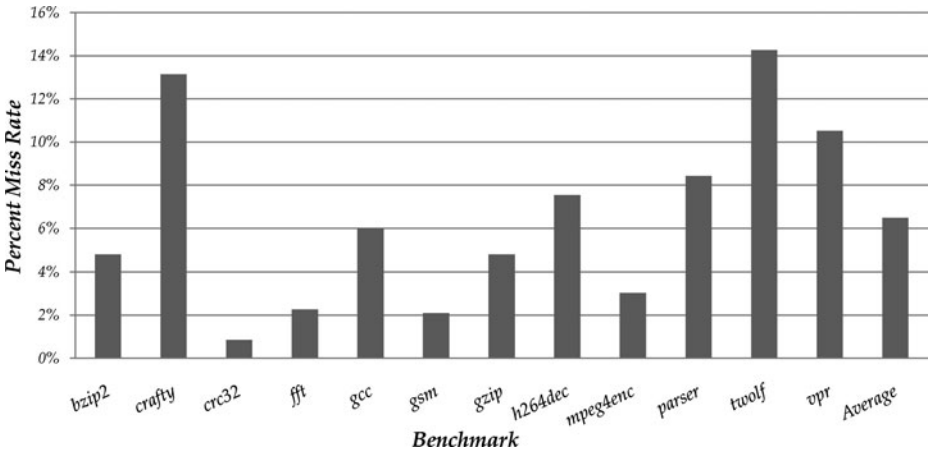

Fig. 12 L1 cache miss rates

Table 2 Baseline execution run-time inefficiency

Benchmark	Baseline		
	Total cycles [10^7]	Dead cycles [10^7]	Inefficiency [%]
bzip2	23,265.90	3,708.52	15.94
crafty	3,992.23	1,270.59	31.83
crc32	55.15	1.81	3.29
fft	67.30	4.34	6.45
gcc	255.01	53.31	20.91
gsm	65.32	1.66	2.54
gzip	12,562.51	1,670.36	13.30
h264dec	1,717.58	388.47	22.62
mpeg4enc	1,161.36	107.22	9.23
parser	1,750.59	407.27	23.26
twolf	1,916.39	596.40	31.12
vpr	1,567.28	496.37	31.67

ory stalls, along with the associated percentage of execution overhead due to memory stalls, and the percentage improvement over the baseline in terms of overall run-time execution. In addition, Fig. 13 graphically shows the percentage improvement in run-time execution behavior using our implementation. The average of the execution time improvement across the benchmarks for our proposed implementation is 11.27%.

In terms of area impact, our proposed augmentations to the hardware are quite minimal. Assuming a 45 nm process technology, the additional storage elements and associated combinational logic account for an area impact of approximately 0.0211 mm^2. Comparing this to the ARM1156T2(F)-S processor, whose die size measures 1.25 mm^2 with the same cache configuration used in simulation, one can see that the area overhead is nominal; the additional hardware equates to an area increase of about 1.69%.

Since we only use a small amount of additional hardware, the power efficiency impact of the proposed technique is also quite small. Overall, the structures proposed account for ap-

Table 3 Overall execution run-time improvement over baseline

Benchmark	Proposed design			
	Total cycles [10^7]	Dead cycles [10^7]	Inefficiency [%]	Improvement [%]
bzip2	20,360.27	817.21	4.01	12.43
crafty	3,040.86	316.45	10.41	23.91
crc32	54.19	0.91	1.67	1.65
fft	63.75	0.80	1.26	5.26
gcc	219.46	17.83	8.12	13.92
gsm	64.49	0.81	1.25	1.30
gzip	11,293.27	399.37	3.54	10.12
h264dec	1,478.33	146.97	9.94	14.07
mpeg4enc	1,091.96	39.45	3.61	5.83
parser	1,507.21	163.41	10.84	13.93
twolf	1,715.91	395.36	23.04	10.50
vpr	1,217.55	146.29	12.02	22.34

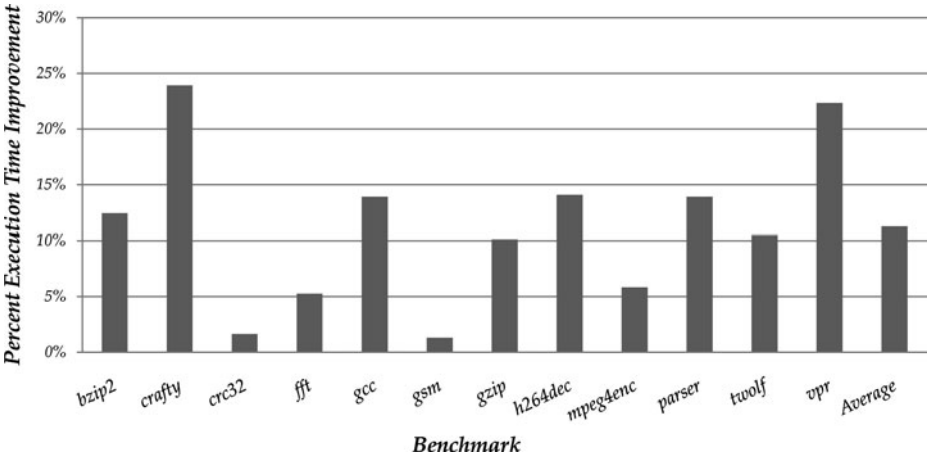

Fig. 13 Overall run-time execution improvement

proximately 2 KB of additional storage elements, along with some necessary routing signals and muxing. Since our implementation does not rely on creating a central register/operand storage area, we do not suffer the same routing and muxing overhead that exponentially grows when dealing with reorder buffers and instruction queues. Furthermore, indexing into our hardware structure only occurs during load instructions, as opposed to every single instruction. Thus, when compared to the complexity of a fully dynamic instruction reordering processor, our implementation is far more efficient. According to [23], a typical out-of-order processor's ROB, register renaming table, and instruction queue on average consume about 27%, 13.5%, and 26% of the total energy usage of the processor, respectively. This combines for a total energy consumption of about 67% of the total processor, a figure that is not feasible when considering the power-conscientious mobile embedded domain. On the other hand, the energy consumption of our proposal is about 0.6% of the total energy footprint for

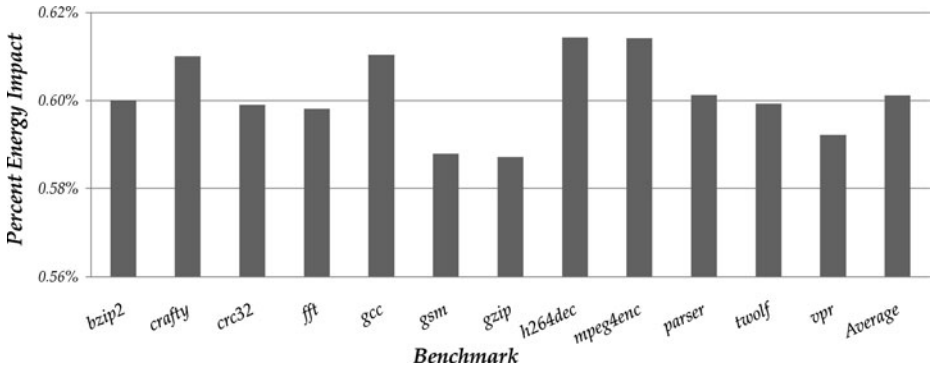

Fig. 14 Overall energy consumption contribution

a comparable in-order processor, using a CACTI [24] approximation as shown in Fig. 14. Therefore, our proposed technique can deliver most of the same benefits with regard to resolving memory stalls, but at a fraction of the energy consumption and area utilization of an out-of-order processor.

6 Conclusions

We have presented a novel architecture for leveraging static compile-time analysis information and hiding the impact of memory stalls by configuring and allowing independent instructions to continue executing during a memory stall at run-time. As shown, memory instructions account for a large proportion of the instructions within an application. Although much work has gone into mitigating cache misses, processors still encounter a substantial amount of overhead when misses do occur. Mobile embedded processors, being far more resource constrained, cannot easily leverage the same technology used in general-purpose machines to overcome memory stalls. In particular, out-of-order execution is prohibitively expensive in terms of area and power utilization in the mobile embedded domain.

By leveraging both compile-time and run-time information, we are able to propose an architecture capable of delivering most of the benefits of out-of-order execution with regard to memory pipeline stalls, but at far less the cost. The key principle is the strengthening of the interaction between the compiler and the underlying hardware microarchitecture, allowing information not readily available during run-time (e.g. global data interdependencies) to be gathered and conveyed to the hardware by the compiler.

The achievement of these goals has been confirmed by extensive experimental results. A significant reduction in the number of dead cycles due to memory stalls has been demonstrated by a representative set of simulation results. By mitigating a large majority of these wasted pipeline cycles during cache misses, we are able to realize substantial improvements in the overall program execution time. The proposed technique has significant implications for mobile embedded processors, especially with regard to high-performance, power-sensitive devices such as cellular smartphones and MP3 players, as it yields improvements to execution time while minimally affecting power consumption. Thus, we can deliver faster performance without negatively affecting battery life.

As portable embedded processors continue to spread and become ubiquitous, it is essential to maintain high performance, low power, and small size. The proposed architecture

fulfills these requirements and enables mobile embedded processors to continue to mature and be able to handle exceedingly complex and aggressive applications, while maintaining their battery life.

References

1. Wilkes MV (2001) The memory gap and the future of high performance memories. SIGARCH Comput Archit News 29(1):2–7
2. Lee L, Kannan S, Fridman J (2004) MPEG4 video codec on a wireless handset baseband system. In: Proc workshop media and signal processors for embedded systems and SoCs
3. Jouppi NP (1990) Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. SIGARCH Comput Archit News 18:364–373
4. Bournoutian G, Orailoglu A (2008) Miss reduction in embedded processors through dynamic, power-friendly cache design. In: DAC'08: proceedings of the 45th annual conference on design automation. ACM, New York, pp 304–309
5. Sprangle E, Carmean D (2002) Increasing processor performance by implementing deeper pipelines. SIGARCH Comput Archit News 30(2):25–34
6. Tomasulo RM (1967) An efficient algorithm for exploiting multiple arithmetic units. IBM J Res Develop 11:25–33
7. Smith JE, Pleszkun AR (1985) Implementation of precise interrupts in pipelined processors. In: ISCA'85: proceedings of the 12th annual international symposium on computer architecture. IEEE Comput Soc, Los Alamitos, pp 36–44
8. Hily S, Seznec A (1999) Out-of-order execution may not be cost-effective on processors featuring simultaneous multithreading. In: HPCA'99: proceedings of the 5th international symposium on high performance computer architecture. IEEE Comput Soc, Los Alamitos, pp 64–67
9. Grossman JP (2000) Cheap out-of-order execution using delayed issue. In: ICCD'00: proceedings of the 2000 IEEE international conference on computer design, pp 549–551
10. Callahan D, Kennedy K, Porterfield A (1991) Software prefetching. In: ASPLOS-IV: proceedings of the 4th international conference on architectural support for programming languages and operating systems. ACM, New York, pp 40–52
11. Klaiber AC, Levy HM (1991) An architecture for software-controlled data prefetching. SIGARCH Comput Archit News 19(3):43–53
12. Mowry TC, Lam MS, Gupta A (1992) Design and evaluation of a compiler algorithm for prefetching. In: ASPLOS-V: proceedings of the 5th international conference on architectural support for programming languages and operating systems. ACM, New York, pp 62–73
13. Badawy A-HA, Aggarwal A, Yeung D, Tseng C-W (2001) Evaluating the impact of memory system performance on software prefetching and locality optimizations. In: ICS'01: proceedings of the 15th international conference on supercomputing. ACM, New York, pp 486–500
14. Baer J-L, Chen T-F (1991) An effective on-chip preloading scheme to reduce data access penalty. In: Supercomputing'91: proceedings of the 1991 ACM/IEEE conference on supercomputing. ACM, New York, pp 176–186
15. Fu JWC, Patel JH, Janssens BL (1992) Stride directed prefetching in scalar processors. In: MICRO 25: proceedings of the 25th annual international symposium on microarchitecture. IEEE Comput Soc, Los Alamitos, pp 102–110
16. Joseph D, Grunwald D (1997) Prefetching using Markov predictors. In: ISCA'97: proceedings of the 24th annual international symposium on computer architecture. ACM, New York, pp 252–263
17. Park S, Shrivastava A, Paek Y (2008) Hiding cache miss penalty using priority-based execution for embedded processors. In: DATE'08: proceedings of the conference on design, automation and test in Europe, pp 1190–1195
18. Olukotun K, Nayfeh BA, Hammond L, Wilson K, Chang K (1996) The case for a single-chip multiprocessor. SIGOPS Oper Syst Rev 30(5):2–11
19. Austin T, Larson E, Ernst D (2002) Simplescalar: an infrastructure for computer system modeling. Computer 35(2):59–67
20. SPEC CPU2000 Benchmarks. http://www.spec.org/cpu/

21. Lee C, Potkonjak M, Mangione-Smith WH (1997) Mediabench: a tool for evaluating and synthesizing multimedia and communications systems. In: MICRO 30: proceedings of the 30th annual ACM/IEEE international symposium on microarchitecture. IEEE Comput Soc, Los Alamitos, pp 330–335
22. Guthaus MR, Ringenberg JS, Ernst D, Austin TM, Mudge T, Brown RB (2001) Mibench: a free, commercially representative embedded benchmark suite. In: WWC'01: proceedings of the IEEE international workshop on workload characterization. IEEE Comput Soc, Los Alamitos, pp 3–14
23. Folegnani D, González A (2001) Energy-effective issue logic. SIGARCH Comput Archit News 29(2):230–239
24. Wilton SJE, Jouppi NP (1996) CACTI: an enhanced cache access and cycle time model. IEEE J Solid-State Circuits 31(5):677–688