Processor-Memory Co-Exploration driven by a Memory-Aware Architecture Description Language*

Prabhat Mishra pmishra@cecs.uci.edu
Peter Grun pgrun@cecs.uci.edu
Nikil Dutt dutt@cecs.uci.edu
Alex Nicolau nicolau@cecs.uci.edu

Architectures and Compilers for Embedded Systems (ACES) Laboratory
Center for Embedded Computer Systems, University of California, Irvine, CA 92697, USA

Abstract

Memory represents a major bottleneck in modern embedded systems. Traditionally, memory organizations for programmable systems assumed a fixed cache hierarchy. With the widening processor-memory gap, more aggressive memory technologies and organizations have appeared, allowing customization of a heterogeneous memory architecture tuned for the application. However, such a processor-memory co-exploration approach critically needs the ability to explicitly capture heterogeneous memory architectures. We present in this paper a language-based approach to explicitly capture the memory subsystem configuration, and perform exploration of the memory architecture to trade-off cost versus performance. We present a set of experiments using our Memory-Aware Architectural Description Language to drive the exploration of the memory subsystem for the TIC6211 processor architecture, demonstrating a range of cost and performance attributes.

1 Introduction

For a large class of embedded systems, memory represents a major performance, cost and power bottleneck [16]. Thus system designers pay great attention to the design and tuning of the memory architecture early in the design process. However, not many system-level tools exist to help the system designers evaluate the effects of novel memory architectures, and facilitate simultaneous exploration of the processor and memory architectures.

While a traditional memory architecture for programmable systems was organized as a cache hierarchy, the widening processor/memory performance gap [18] requires more aggressive use of memory configurations, *customized* for the specific target applications. To address this problem, on one hand recent advances in memory technology have generated a plethora of new and efficient memory modules (e.g., SDRAM, DDRAM, RAMBUS, etc.), exhibiting a heterogeneous set of features (e.g., page-mode, burst-mode, pipelined accesses). On the other hand, many embedded applications exhibit varied memory access patterns that naturally map into a range of heterogeneous memory configurations (containing for instance multiple cache hierarchies, stream buffers, and on- and off-chip direct mapped memories). In the design of traditional programmable systems, the processor architect typically assumed a fixed cache hierarchy, and spent significant amounts of time optimizing the processor architecture; thus the memory architecture is implicitly fixed (transparent to the processor) and optimized separately from the processor architecture. Due to the heterogeneity in recent memory organizations and modules, there is a critical need to address the memory-related optimizations simultaneously with the processor architecture and the target application. Through co-exploration of the processor and the memory architecture, it is possible to better exploit the heterogeneity in the memory subsystem organizations, and better trade-off system attributes such as cost, performance, and power. However, such processor-memory co-exploration requires the capability to explicitly capture, exploit, and refine both the processor as well as the memory architecture.

Recent work on language-driven Design Space Exploration (DSE) ([1], [3], [4], [6], [8], [12], [17], [21], [22]), uses Architectural Description Languages (ADL) to capture the processor architecture, generate automatically a software toolkit (including compiler, simulator, assembler) for that processor, and provide feedback to the designer on the quality of the architecture. While these approaches extensively address processor features (such as instruction set, number of functional units, etc.) to our knowledge no previous approach allows explicit capture of a customized, heterogeneous memory architecture, and the attendant tasks of generating a software toolkit that fully exploits this memory architecture.

The contribution of this paper is the explicit description of a customized, heterogeneous memory architecture in our EXPRESSION ADL [12], permitting co-exploration of the processor and the memory architecture. By viewing the

*This work was partially supported by grants from NSF (MIP-9708067), DARPA (F33615-00-C-1632) and a Motorola fellowship.

memory subsystem as a "first class object", we generate a memory-aware software toolkit (compiler and simulator), and allow for memory-aware Design Space Exploration (DSE).

The rest of the paper is organized as follows. Section 2 presents related work addressing ADL-driven DSE approaches. Section 3 outlines our approach and the overall flow of our environment. Section 4 presents the memory subsystem description in EXPRESSION, followed by a contemporary example architecture in Section 5. Section 6 illustrates memory architecture exploration using experiments on the TIC6211 processor, with varying memory configurations to trade-off cost versus performance. Section 7 concludes the paper.

2 Related Work

An extensive body of recent work addresses Architectural Description Language (ADL) driven software toolkit generation and Design Space Exploration (DSE) for processor-based embedded systems, in both academia: ISDL [4], Valen-C [5], MIMOLA [6], LISA [7], nML [8], [17], and industry: ARC [1], Axys [2], RADL [19], Target [20], Tensilica [21], MDES [22].

While these approaches explicitly capture the processor features to varying degrees (e.g., instruction set, structure, pipelining, resources), to our knowledge, no previous approach has explicit mechanisms for specification of a customized memory architecture that describes the specific types of memory modules (e.g., caches, stream/prefetch buffers), their complex memory features (e.g., page-mode, burst-mode accesses), their detailed timings, resource utilization, and the overall organization of the memory architecture (e.g., multiple cache hierarchies, partitioned memory spaces, direct-mapped memories, etc.)

Our EXPRESSION ADL [12] was designed to explicitly capture the memory architecture, representing both the structure, and the instruction set of the memory subsystem. The structure represents the connectivity between the memory modules, the characteristics of the memory modules (such as timing), pipelining, and parallelism present. The memory instruction set contains the load/store instructions, cache control, DMA, prefetch instructions, as well as the operations triggered automatically by the hardware (such as cache line fill transfer from the main memory).

We use the detailed memory subsystem information in EXPRESSION to automatically generate EXPRESS, our Memory-Aware Compiler ([9], [10]) to better match the memory subsystem architecture (by using the explicit resource and timing information in scheduling to hide the latency of the lengthy memory operations), and generate SIMPRESS [14], the cycle-accurate structural memory simulator to provide accurate feedback to the designer. The explicit memory subsystem ADL specification enables processor-memory co-exploration, leading to a larger design space, better fine-tuning of the memory subsystem to the processor architecture and application, and better performance/cost trade-offs.

3 Our Approach

Figure 1 shows the flow in our approach. In our IP library based Design Space Exploration (DSE) scenario, the designer starts by selecting a set of components from a processor IP library and memory IP library. Our EXPRESSION Architectural Description Language (ADL) description (containing a mix of such IP components and custom blocks) is then used to generate the information necessary to target both the compiler and the simulator to the specific processor-memory system.

Traditionally, the memory subsystem was transparent (assumed an implicitly defined memory architecture, e.g., a fixed cache hierarchy) to the processor and the software toolkit. While the processor pipeline was captured in detail to allow aggressive scheduling in the compiler, the memory subsystem pipeline was not explicitly captured and exploited by the compiler. However, by describing the pipelining and parallelism available in recent memory organizations, there is tremendous opportunity for the compiler to generate performance improvements. Indeed, as shown in Figure 1, our previous work on RTGEN [11] (Reservation Tables generation algorithm) and TIMGEN [9] (Timing Generation algorithm) already generates the timing information for both the processor and memory subsystem pipelines starting from the ADL description of the memory. The compiler uses this detailed timing information to hide the latency of the lengthy memory operations in the presence of efficient memory access modes (e.g., page/burst modes), and cache hierarchies [10], to generate significant performance improvements. Such aggressive optimizations are only possible due to the explicit representation of the detailed memory architecture.

We present here the memory subsystem description in EXPRESSION, along with the abstractions that allow capturing a set of heterogeneous memory modules, and connecting them to form customized memory architectures. Furthermore, in a DSE environment it is crucial to provide the designer with detailed feedback on the choices made in the processor and memory architectures. In [14] we presented SIMPRESS, our cycle-accurate structural simulator generation approach for the processor descriptions in EXPRESSION. In this paper we present the memory simulator generation (shown shaded in Figure 1) that is integrated into the SIMPRESS simulator, allowing for detailed feedback on the memory subsystem architecture and its match to the target applications.

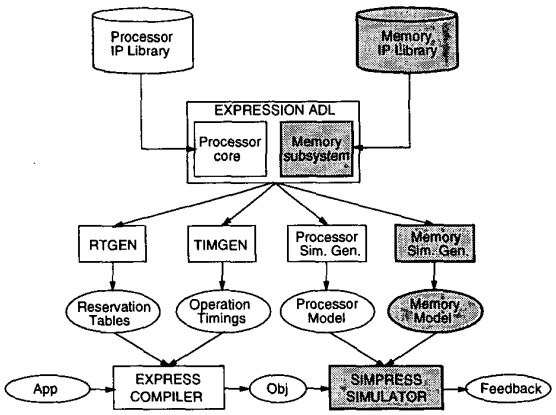

Figure 1. The Flow in our approach

4 The Memory Subsystem Description in EXPRESSION

In order to explicitly describe the memory architecture in EXPRESSION, we need to capture: (i) the behavior of the operations that touch the memory subsystem (the memory subsystem instruction set description), and (ii) the organization of the memory subsystem (the memory subsystem structure).

The memory subsystem instruction set represents the possible operations that can occur in the memory subsystem, such as data transfers between different memory modules or to the processor, control instructions for the different memory components (such as the DMA), or explicit cache control instructions (e.g., cache freeze, etc.).

The memory subsystem structure represents the abstract memory modules (such as caches, stream buffers, RAM modules), their connectivity and characteristics (e.g., cache properties). The memory subsystem structure is represented as a netlist of memory components connected with each other and with the processor through connections and ports. The memory components are described and attributed with their characteristics (such as cache line size, replacement policy, write policy).

The pipeline stages and parallelism for each memory module, its connections and ports, as well as the latches between the pipeline stages are described explicitly, to allow modeling of resource and timing conflicts in the pipeline. The semantics of each component is represented in C, as part of a parameterizable components library.

5 Example Memory Architecture

We illustrate our Memory-Aware Architectural Description Language (ADL) using the Texas Instruments TIC6211 VLIW DSP processor that has several novel memory features. Figure 2 shows the example architecture, containing an off-chip DRAM, an on-chip SRAM, and two levels of cache (L1 and L2), attached to the memory controller of the TIC6211 processor For illustration purposes we present only the D1 ld/st functional unit of the TIC6211 processor, and we omitted the External Memory Interface unit from the Figure 2. TI C6211 is an 8-way VLIW DSP processor with a deep pipeline, composed of 4 fetch stages (PG, PS, PR, PW), 2 decode stages (DP, DC), followed by the 8 functional units. The D1 load/store functional unit pipeline is composed of D1_E1, D1_E2, and the 2 memory controller stages: MemCtrl_E1 and MemCtrl_E2.

The L1 cache is a 2-way set associative cache, with a size of 64 lines, a line size of 4 words, and word size of 4 bytes. The replacement policy is Least Recently Used (LRU), and the write policy is write-back. The cache is composed of a TAG_BLOCK, a DATA_BLOCK, and the cache controller, pipelined in 2 stages (L1_S1, L1_S2). The cache characteristics are described as part of the STORAGE_SECTION in EXPRESSION [12]:

```
(L1_CACHE
  (TYPE DCACHE)
  (NUM_LINES 64)
  (LINESIZE 4)
  (WORDSIZE 4)
  (ASSOCIATIVITY 2)
  (REPLACEMENT_POLICY LRU)
  (WRITE_POLICY WRITE_BACK)
  (SUB_UNITS TAG_BLOCK DATA_BLOCK L1_S1 L1_S2)
)
```

The memory subsystem instruction set description is represented as part of the Operation Section in EXPRESSION [12]:

```
(OPCODE LDW (OPERANDS (SRC1 reg) (SRC2 reg) (DST reg))
```

The internal memory subsystem data transfers are represented explicitly in EXPRESSION as operations. For instance, the L1 cache line fill from L2 triggered on a cache miss is represented through the LDW_L1_MISS operation, with the memory subsystem source and destination operands described explicitly:

```
(OPCODE LDW_L1_MISS (OPERANDS (SRC1 reg)
(SRC2 reg) (DST reg) (MEM_SRC1 L1_CACHE)
(MEM_SRC2 L2_CACHE) (MEM_DST1 L1_CACHE))
```

This explicit representation of the internal memory subsystem data transfers (traditionally not present in ADLs) allows the designer to reason about the memory subsystem configuration. Furthermore it allows the compiler to exploit the organization of the memory subsystem, and the simulator to provide detailed feedback on the internal memory subsystem traffic. We do not modify the processor instruction set, but rather represent explicitly operations which are implicit in the processor and memory subsystem behavior.

The pipelining and parallelism between the cache operations is described in

EXPRESSION through PIPELINE_PATHS [12]. Pipeline Paths represent the ordering between pipeline stages in the architecture (represented as bold arrows in Figure 2). For instance, a load operation to a DRAM address traverses first the 4 fetch stages (PG, PS, PR, PW) of the processor, followed by the 2 decode stages (DP, DC), and then it is directed to the load/store unit D1. Here it traverses the D1_E1 and D1_E2 stages, and is directed by the MemCtrl_E1 stage to the L1 cache, where it traverses the L1_S1 stage. If the access is a hit, it is then directed to the L1_S2 stage, and the data is sent back to the MemCtrl_E1 and MemCtrl_E2 (to keep the figure simple, we omitted the reverse arrows bringing the data back to the CPU). Thus the pipeline path traversed by the example load operation is:

```
(PIPELINE PG, PS, PR, PW, DP, DC, D1_E1, D1_E2,
 MemCtrl_E1, L1_S1, L1_S2, MemCtrl_E1, MemCtrl_E2)
```

Even though this example pipeline path is flattened, the pipeline paths in EXPRESSION are described in a hierarchical manner. In case of an L1 miss, the data request is redirected from L1_S1 to the L2 cache controller, as shown by the pipeline path (the bold arrow) to L2 in Figure 2.

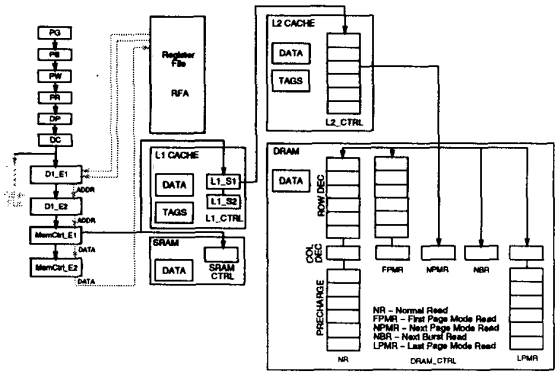

Figure 2. Sample Memory Architecture for TIC6211

The L2 cache is 4-way set associative, with a size of 1024 lines, and line size of 8 words. The L2 cache controller is non-pipelined, with a latency of 6 cycles:

```
(L2_CTRL (LATENCY 6))
```

During the third cycle of the L2 cache controller, if a miss is detected it is sent to the off-chip DRAM. The DRAM module is composed of the DRAM data block and the DRAM controller, and supports normal, page-mode and burst-mode accesses. A normal access starts with a row decode, where the row part of the address is used to select a particular row from the data array, and copy it into the row buffer. During the column decode, the column part of the address is used to select a particular element from the row buffer and output it. During the precharge, the bank is deactivated. In a page-mode access, if the next access is to the same row, the data can be fetched directly form the row buffer, omitting the column decode and precharge operations. During a burst access, consecutive elements from the row buffer are clocked out on consecutive cycles. Both page-mode and burst-mode accesses, when exploited judiciously generate substantial performance improvements [9]. The timings of each such access mode is represented using the pipeline paths and LATENCY constructs. For instance, the normal read access (NR), composed of a column decode, a row decode and a precharge, is represented by the pipeline path:

```
(PIPELINE COL_DEC ROW_DEC PRECHARGE)
...
(COL_DEC (LATENCY 6))
(ROW_DEC (LATENCY 1))
(PRECHAREGE (LATENCY 6))
```

where the latency of the COL_DEC is 6 cycles, of ROW_DEC is 1 cycle, and of the PRECHARGE is 6 cycles.

In this manner EXPRESSION can model a variety of memory modules and their characteristics. A unique feature of EXPRESSION is the ability to model the *parallelism* and *pipelining* available in and between the memory modules, such as number of outstanding hits, misses or parallel loads, and generate timing and resource information to allow aggressive scheduling to hide the latency of the lengthy memory operations. The EXPRESSION description can be used to drive the generation of both a memory-aware compiler [9], [10], and cycle-accurate structural memory subsystem simulator, and thus enable Design Space Exploration and co-design of the memory and processor architecture. For more details on the memory subsystem description in EXPRESSION and automatic simulator generation, please refer to [15].

6 Experiments

As described earlier, we have already used this Memory-Aware Architectural Description Language (ADL) approach to generate a Memory-Aware Compiler [9] and manage the memory miss traffic [10], resulting in significantly improved performance. In this section we demonstrate further use of the memory subsystem specification to describe different configurations of the memory subsystem with the goal of studying the trade-off between cost and performance.

6.1 Experimental Setup

We performed a set of experiments starting from the base TIC6211 processor architecture, and varied the memory subsystem architecture. We generated a cycle-accurate structural memory subsystem simulator, and performed Design Space Exploration of the memory subsystem. The memory organization of the TIC6211 is varied by using an

L1 cache, L2 cache, an off-chip DRAM module, an on-chip SRAM module and a stream buffer module [13]. The L1 cache is a 2-way set associative cache with line size of 4 words and word of 4 bytes. The L2 cache shares a total of 2K on-chip SRAM memory with the direct mapped on-chip SRAM.

The Stream Buffer [13] is used as a replacement for the L2 cache, exhibiting a much smaller data storage size, and slightly more complex control mechanism. It receives a sequence of miss addresses from L1, storing them into a small history buffer. When it recognizes a stream, it allocates one of several FIFO queues to start prefetching it from the DRAM. The stream buffer we implemented contains 4 such FIFO queues, storing 4 cache lines each, and uses an LRU policy to discard a FIFO in case of conflict. When the stream buffer receives an L1 cache miss address, it compares it to the top of each FIFO queue. If the address is found - a stream buffer hit - it pops it from the FIFO and returns it to the L1 cache. If the address is not found - a miss - the stream buffer compares it with the addresses in the history buffer to check for a stream, and sends a request to the DRAM to bring the data.

We used a set of benchmarks from the multimedia and DSP domains, and compiled them using the EXPRESS compiler. We collected the statistics information using the SIMPRESS cycle-accurate structural simulator, which models both the TI6211 processor and the memory subsystem.

6.2 Results

The configurations we experimented with are presented in Table 1. The numbers in Table 1 represent: the size of the memory module (e.g., the size of L1 in configuration 1 is 128), the cache/stream buffer organizations: $num\_lines \times num\_ways \times line\_size \times word\_size$, the latency (in number of processor cycles), and the replacement policy (LRU or FIFO). Note that for the stream buffer, num_ways represents the number of FIFO queues present.

Table 1. The memory subsystem configurations

Config	L1 Cache	L2 Cache	SRAM	Stream Buffer	DRAM
1	4x2x4x4 lat=1 (LRU)	-	-	4x4x4x4 lat=4	lat=20 cycle
2	4x2x4x4 lat=1 (LRU)	-	2K lat=1	-	lat=20 cycle
3	4x2x4x4 lat=1 (FIFO)	16x4x8x4 lat=4 (FIFO)	-	-	lat=20 cycle
4	4x2x4x4 lat=1 (LRU)	16x4x8x4 lat=4 (LRU)	-	-	lat=20 cycle
5	4x2x4x4 lat=1 (FIFO)	32x1x8x4 lat=4 (FIFO)	1K lat=1	-	lat=20 cycle
6	-	-	8K lat=1	-	lat=20 cycle

The configurations in Table 1 are presented in increasing order of the cost in terms of area. The first configuration contains an L1 cache and a small stream buffer (256 bytes) to capitalize on the stream nature of the benchmarks. The second configuration contains the L1 cache and an on-chip direct mapped SRAM of 2K. A part of the arrays in the application are mapped to the SRAM. Due to the reduced control necessary for the SRAM, it has a small latency (of 1 cycle), and the area requirements are small. The third configuration contains L1 and L2 caches with FIFO replacement policy. Due to the control necessary for the L2 cache (of size 2K), the cost of this configuration is larger than the configuration 2 containing the SRAM. Configuration 4 is the same as configuration 3, but with LRU replacement policy for the L1 and L2 caches. Due to the more complex control required by the LRU policy, the cost of this configuration is larger than configuration 3. Configuration 5 contains an L1 cache, an L2 cache of size 1K and a direct mapped SRAM of size 1K. Due to the extra busses to route the data to the caches and SRAM, this configuration has a larger cost than the previous one. The last configuration contains a large SRAM, and the caches, and has the largest area requirement. All the configurations contain the same off-chip DRAM module with a latency of 20 cycles.

Figure 3 presents a subset of experiments we ran, showing the total cycle counts (including the time spent in the processor) for the set of benchmarks for different memory configurations attached to the TIC6211 processor. From the experiments we performed, we chose a representative set of benchmarks, which show the different trends in the cost versus performance trade-off. Even though these benchmarks are kernels, we observed a significant variation in the trends shown by the different applications.

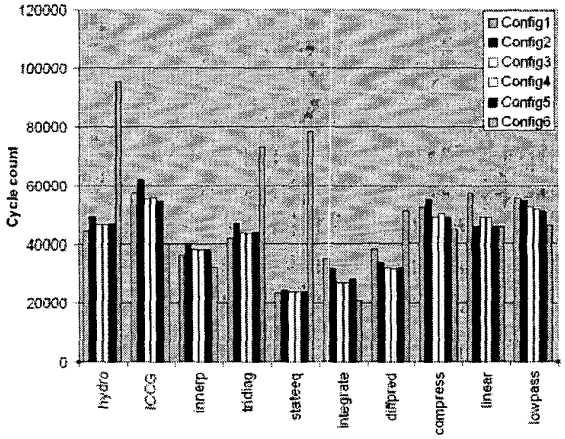

Figure 3. Cycle counts for the memory configurations

For instance, in hydro, tridiag, and stateeq the first configuration even though has the lowest cost performs the best (lower cycle count means higher performance), due to the capability of the stream buffer to exploit efficiently the stream nature of the access patterns. Moreover, in these ap-

plications the most expensive configuration (configuration 6), containing the large SRAM behaves poorly, due to the fact that not all the arrays fit in the SRAM, and the lack of L1 cache to compensate the large latency of the DRAM creates its toll on the performance.

The expected trend of higher cost - higher performance was apparent in the applications ICCG, integrate, and low-pass, While the stream buffer in configuration 1 has a comparable performance to the other configurations, the configuration 6 has the best behavior due to the low latency of the direct mapped on-chip SRAM.

Thus, using our Memory-Aware ADL-based Design Space Exploration approach, we obtained design points with varying cost and performance. We observed various trends for different application classes, allowing customization of the memory architecture tuned to the applications. Note that this cannot be determined through analysis alone; the customized memory subsystem must be explicitly captured, and the applications have to be executed on the configured processor-memory system, as we demonstrated in this section.

7 Summary

Memory represents a critical driver in terms of cost, performance and power for embedded systems. To address this problem, a large variety of modern memory technologies, and heterogeneous memory organizations have been proposed.

On one hand the application is characterized by a variety of access patterns (such as stream, locality-based, etc.). On the other hand, new memory modules and organizations provide a set of features which exploit specific applications needs (e.g., caches, stream buffers, page-mode, burst-mode, DMA). To find the best match between the application characteristics and the memory organization features, the designer needs to explore different memory configurations in combination with different processor architectures, and evaluate each such system for a set of metrics (such as cost, power, performance). Performing such processor-memory co-exploration requires the capability to capture the memory subsystems, and perform a compiler-in-the-loop exploration/evaluation.

In this paper we presented a Memory-Aware Architectural Description Language (ADL) approach which captures the memory subsystem explicitly.

This Memory-Aware ADL approach is used to drive the generation of a cycle accurate memory simulator, and also facilitate the exploration of various memory configurations, and trade-off cost versus performance. Our experimental results show that varying price-performance design points can be uncovered using the processor-memory co-exploration approach.

Our ongoing work targets the use of this ADL approach for further memory exploration experiments, using larger applications, to study the impact of different parts of the application (such as loop nests) on the memory organization behavior and overall performance, as well as on system power.

References

[1] ARC Cores. *http://www.arccores.com.*

[2] Axys Design Automation. *http://www.axysdesign.com.*

[3] G. G. et al. CHESS: Retargetable code generation for embedded DSP processors. In *Code Generation for Embedded Processors.* Kluwer, 1997.

[4] G. H. et al. ISDL: An instruction set description language for retargetability. In *Proc. DAC*, 1997.

[5] H. Y. et al. A programming language for processor based embedded systems. In *Proc. APCHDL*, 1998.

[6] R. L. et al. Retargetable generation of code selectors from HDL processor models. In *Proc. EDTC*, 1997.

[7] V. Z. et al. LISA - machine description language and generic machine model for HW/SW co-design. In *IEEE Workshop on VLSI Signal Processing*, 1996.

[8] M. Freericks. The nML machine description formalism. Technical Report TR SM-IMP/DIST/08, TU Berlin CS Dept., 1993.

[9] P. Grun, N. Dutt, and A. Nicolau. Memory aware compilation through accurate timing extraction. In *DAC*, Los Angeles, 2000.

[10] P. Grun, N. Dutt, and A. Nicolau. Mist: An algorithm for memory miss traffic management. In *To Appear in ICCAD*, San Jose, 2000.

[11] P. Grun, A. Halambi, N. Dutt, and A. Nicolau. RTGEN: An algorithm for automatic generation of reservation tables from architectural descriptions. In *ISSS*, San Jose, CA, 1999.

[12] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. EXPRESSION: A language for architecture exploration through compiler/simulator retargetability. In *Proc. DATE*, Mar. 1999.

[13] N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *ISCA*, 1990.

[14] A. Khare, N. Savoiu, A. Halambi, P. Grun, N. Dutt, and A. Nicolau. V-SAT: A visual specification and analysis tool for system-on-chip exploration. In *Proc. EUROMICRO*, 1999.

[15] P. Mishra, P. Grun, N. Dutt, and A. Nicolau. Memory subsystem description in EXPRESSION. Technical report, University of California, Irvine, 2000.

[16] S. Przybylski. Sorting out the new DRAMs. In *Hot Chips Tutorial*, Stanford, CA, 1997.

[17] V. Rajesh and R. Moona. Processor modeling for hardware software codesign. In *International Conference on VLSI Design*, Jan. 1999.

[18] Semiconductor Industry Association. *National technology roadmap for semiconductors: Technology needs*, 1998.

[19] C. Siska. A processor description language supporting retargetable multi-pipeline dsp program development tools. In *Proc. ISSS*, Dec. 1998.

[20] Target Compiler Technologies. *http://www.retarget.com.*

[21] Tensilica Incorporated. *http://www.tensilica.com.*

[22] Trimaran Release: http://www.trimaran.org. *The MDES User Manual*, 1997.