SoMMA: A software-managed memory architecture for multi-issue processors

Tiago Trevisan Jost, Gabriel Luca Nazar, Luigi Carro

Version of Record: https://www.sciencedirect.com/science/article/pii/S0141933120303069
Manuscript_a190c95f12503915a6e1a5687f943206

SoMMA: A Software-managed Memory Architecture for Multi-issue Processors

Tiago Trevisan Jost[1], Gabriel Luca Nazar[2], and Luigi Carro[2]

[1] CEA, LETI, Univ. Grenoble Alpes, 38000, Grenoble, France
[2] Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil

Abstract— Embedded processors rely on the efficient use of instruction-level parallelism to answer the performance and energy needs of modern applications. Though improving performance is the primary goal for processors in general, it might lead to a negative impact on energy consumption, a particularly critical constraint for current systems. In this paper, we present SoMMA, a software-managed memory architecture for embedded multi-issue processors that can reduce energy consumption and energy-delay product (EDP), while still providing an increase in memory bandwidth. We combine the use of software-managed memories (SMM) with the data cache, and leverage the lower energy access cost of SMMs to provide a processor with reduced energy consumption and EDP. SoMMA also provides a better overall performance, as memory accesses can be performed in parallel, with no cost in extra memory ports. Compiler-automated code transformations minimize the programmer's effort to benefit from the proposed architecture. The approach shows average speedups of 1.118x and 1.121x, while consuming up to 11% and 12.8% less energy when comparing two modified ρVEX processors and their baselines, at full-system level comparisons. SoMMA also shows reduction of up to 41.5% on full-system EDP, maintaining the same processor area as baseline processors.

Index Terms— code generation process, instruction-level parallelism, memory bandwidth limitation, multi-issue processors, software-managed memory

1 Introduction

TECHNOLOGY scaling has allowed computer architects to exponentially increase the frequency of processors for many years [1]. Due to power restrictions, however, the hardware industry was forced to switch to a multi-processor approach, demanding the usage of even higher memory bandwidth. However, memory manufacturers did not achieve the same order of speed improvement, becoming one of the bottlenecks for increasing the overall performance of a system. This adverse scenario led to a great effort on developing smart solutions to the limited memory bandwidth problem. Such solutions include the exploration of temporal and spatial locality through memory hierarchy, and instruction and data prefetching [2], [3]. Although improvements have been made, the gap between processor and memory performance is still significant. Thus, the concept of "Memory Wall" [4], which states that the rate of improvement in DRAM speed does not follow the improvement in processor speed, became a challenge for the community in general.

Processors exploiting parallelism, either in instruction or thread level, may need to perform multiple accesses in a single clock cycle to maintain parallelism, and thus, multiple memory ports could be used to achieve that goal. Even when exploiting access locality through cache memories to minimize this limitation, multi-ported memories are known to introduce significant costs. The study in [5] shows that the impact of increasing the number of ports in a memory system leads to a quadratic increase in the cell area. Additionally, extra memory ports would also implicate on higher dynamic and static energy consumption, which are especially undesirable for embedded systems, as they are often more power- and energy-constrained than general-purpose computing platforms.

Fig. 1 illustrates leakage current, access time, dynamic and static energy for 32 KB cache memories with 1, 2, 8, and 16 ports, in a 65 nm technology obtained with the Cacti-p tool [6]. We can observe how power and energy attributes escalate in multi-ported memories. For instance, the 16-port cache showed dynamic and static energy increases of more than 7x and 16x, respectively, and even when considering a 2-ported cache, an increase of 25% in static energy might be crucial for embedded processors. Even an access time increase of 15% in a 2-ported cache can possibly compromise energy consumption, as it may increase applications' execution time.

On the other hand, many applications still struggle with memory bandwidth and have their parallelism limited by the number of ports on the cache. The presence of extra ports can potentially improve instruction-level parallelism (ILP) on applications, and therefore, accelerate their execution. For instance, image and video processing applications usually offer high parallelism opportunities due to the sparse of control-flow instructions. Fig. 2 demonstrates the speedup of a set of applications, namely, a fourier inverse transform (itver2), three matrix multiplication with different matrix sizes (m10x10, m16x16 and m32x32), the sum of absolute differences (SAD), and an x264 video encoder, in a very-long instruction word (VLIW) processor when one uses extra cache ports. These results illustrate how the additional ports can have a huge impact on the performance of applications, with some applications being over 3x faster in a 16-ported system.

Due to the end of Dennard scaling [7], concerns over energy consumption play a major role in processor design. Techniques to improve performance might have a negative impact on energy consumption, thus the energy-delay product (EDP) comes into play as a metric to evaluate the quality of a system, combining the influence of performance and energy. When considering EDP, improvements in performance would not justify the high cost of energy consumption shown in Fig. 1 for the set of applications in Fig. 2.

In this work, we present SoMMA, a software-managed memory architecture for embedded multi-issue processors that can reduce energy consumption and energy-delay product (EDP). SoMMA also provides an increase in memory bandwidth, but not using multi-ported memories. Its main characteristic is the replacement of a larger cache by a combination of a smaller cache and a set of software-managed memories (SMMs) in such a way that area remains equivalent in

both processors with and without the SMMs. We use a compiler-based approach to manage the extra memories and accelerate the execution of applications. Previous works like [8] and [9] proposed to use software-managed memories, also called software-controlled memories or scratchpads, as a replacement for caches aiming at reducing energy consumption; others [10], [11] focus on boosting the performance of embedded processors due to the reduction of cache misses. A major advantage of SoMMA in comparison to previous strategies relies on the usage of multiple memories in parallel, inducing an increase on the total bandwidth available and creating new opportunities for ILP exploitation and energy savings. Other approaches mainly obtain gains in execution time due to avoidance of cache misses, while we also cover that, we go further by exploiting ILP through parallel accesses to memories.

SoMMA also differs on how address spaces are treated. Scratchpads from previous works typically use the same address space of the memory hierarchy, i.e., a range of addresses is scratchpad-addressable, while others are dealt through the cache. On the other hand, our memories have unique address spaces, with no correlation among them whatsoever. In many cases the compiler can identify the address space being accessed by each instruction, allowing the parallelization of accesses performed to different spaces, which is crucial to the energy savings we can achieve.

Although SoMMA relies mostly on compiler-automated code transformation rather than major hardware-level modifications, we propose a solution that incorporates changes in both hardware and software to overcome the limit of single-port systems, providing a higher throughput. We demonstrate the efficiency of SoMMA in a VLIW processor, although it could be adapted to other ILP-capable architectures, e.g., superscalar processors. Modifications embedded in the LLVM compiler toolchain [12] eliminate the need to manually manage the modified memory architecture. SoMMA can be used for application-specific integrated circuits (ASIC) processors and those built on the top of field-programmable gate arrays (FPGA), or softcores, as they show the same concerns regarding bandwidth limitation.

This work is organized as follows. Section 2 presents a background on scratchpads and software-managed memory concepts, exploring the main previous works. We also cover

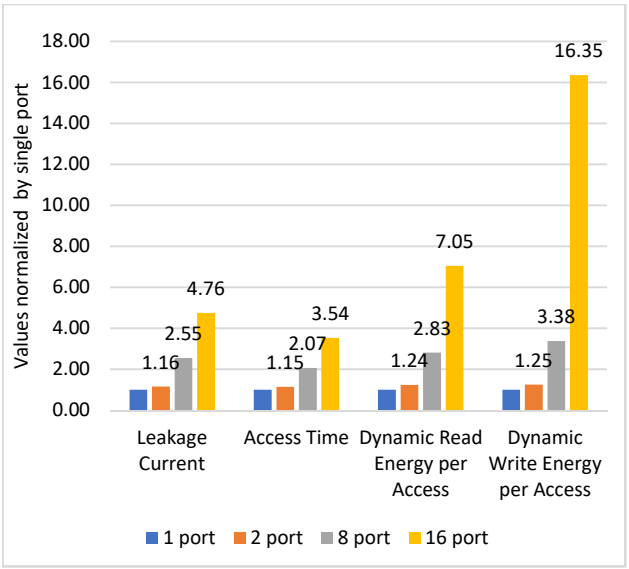

Fig. 1. 32 KB cache memories with different number of ports. Values were normalized over single port. Latency and energy attributes increase significantly in multi-ported caches.

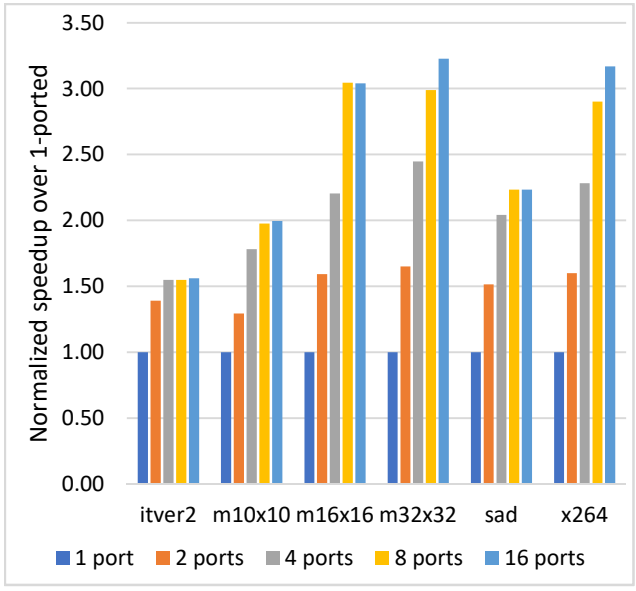

Fig. 2. Normalized Speedup on a VLIW processor. Some applications struggle with limited memory bandwidth. By increasing the number of memory ports, applications on multi-ported systems can perform considerably better than on single-ported.

previous techniques that improve bandwidth through more efficient multi-ported systems. In Section 3, we present an example application that illustrates the approach. SoMMA is described in Section 4, in which we give the hardware and software requirements, explaining the hardware changes on the processor side and the main component: the automated code generation for SoMMA. The experimental results carried out in this work are presented in Section 5. The end of this section illustrates how our LLVM-based code generation performs compared to the original HP compiler for the targeted ISA [13], showing we leverage the myriad of compiler optimizations presented in the infrastructure to overcome the original compiler for the ISA in our set of benchmarks. Conclusions drawn from this work are encountered in Section 6, as well as the future works envisioned from it.

2 Background and Related Work

2.1 Scratchpad-based Systems

Scratchpads, here also referred as software-managed memories, benefit from a simple design with neither tag arrays nor comparators, so they provide significant area and latency improvements over caches. On the other hand, the hardware-level control of caches gives a more transparent way of managing values in memory, as control is taken away from the software.

The two most important schemes for allocating data into software-managed memories are: the static and overlay-based approach [14], [15], [16]. In the static-based approach, data is loaded once at the beginning of the program and remain invariant during its entire execution. Due to its simplicity, this method limits the usability of the memories considerably, as other data cannot be placed at that same location later. When one considers that applications may use data only for a small part of their execution time, a static approach will not be efficient for them. Overlay-based approaches, on the other hand, tackle most of the inefficiency of the static-based methods. The content of the software-managed memory changes dynamically during the execution of the program. This approach offers a flexibility that is absent from the static counterpart, as different data can be stored in the same location depending on the program point.

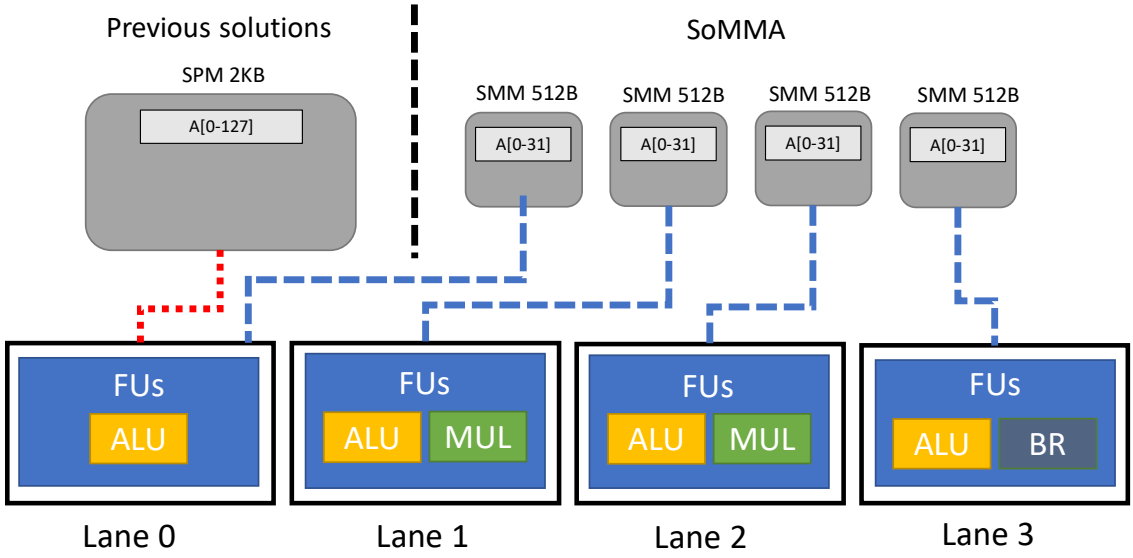

Fig. 3. A 4-issue VLIW processor example. Previous solutions are represented by the red-dotted connection between lane 0 and scratchpad (SPM), while the proposed solution is represented by the blue-dashed connection between software-managed memories (SMMs) and lanes. Our solution uses memories in parallel in order to leverage the multi-issue characteristic of the processor. We can load and store values in multiple memories at once.

Several studies propose the use of scratchpads as a replacement for caches. Verma and Marwedel [16] present an overlay-based method to dynamically copy both variables and code segments onto a scratchpad at runtime. The proposed solution is keen to the Global Register Allocation problem, in which the compiler attempts to make virtual register assignments to physical registers to a minimum, in order to prevent spills to memory. Angiolini et al. [10] used an approach for the placement of instructions into scratchpads, requiring no access to either the compiler or application source code.

The work in [11] proposed a strategy to automatically partition global and stack data among different heterogeneous memory units in embedded systems that lack caching hardware. Udayakumaran et al. [17] shows a complete scheme to allocate data and program code to scratchpads dynamically. The solution handles global variables and stack data, including both segments of data and code at a unique scheme. Horro et al. [18] proposed a technique for multicore processors that places reused variables in scratchpads. The technique places variables in single scratchpads and aims at reducing the dynamic energy consumption of read and write accesses and execution time of applications. Moreover, scratchpads have also been explored in many-core processors [19], [20], and multiprocessor System-on-Chips [21] with new allocation strategies to improve performance and energy consumption.

The industry has also played a major role to advance the use of scratchpad-based systems. The IBM cell processor [22], that powers Sony PlayStation 3 console, uses scratchpad memories inside its processing elements. The architecture aims at accelerating media and streaming workloads by reducing memory latency inside the SPE elements. Che and Chatha [23], on the other hand, use the same IBM Cell processor to accelerate stream applications by placing their code into the scratchpad with an overlay-based method. The work in [14] presents a memory optimization scheme to minimize the usage of shared on-chip memory, in the NVIDIA G80 graphics processing unit (GPU). The optimization is particularly designed to work in an architecture which scratchpads are shared among many threads.

2.2 Multi-ported Systems

The limiting bandwidth in the memory system might also be overcome with the addition of memory ports. Because multi-ported systems can be quite costly, previous works propose techniques to design multi-ported systems more efficiently.

The works [24][25][26] propose techniques to design multi-ported systems more efficiently. Some work focus on FPGA devices, where memory units, called block RAMs (BRAM), are typically dual-ported. Malazgirt et al. [24] presents an application-specific methodology that analyzes a sequential code, extracts parallelism and determines the number of reads and writes ports necessary for such application. Laforest and Steffan [26] introduces a new approach for creating multi-ported memories in FPGAs that efficiently combines BRAMs while achieving significant performance improvements over other methods. The work in [25] applies to ASIC devices and proposes an area and energy-efficient multi-port cache memory. Although able to minimize the costs of multiple ports, it is not able to completely mitigate them.

In [27], the SMM-based approach used in this work is first presented, but it is evaluated considering solely the energy consumption of the on-chip memories. Fig. 3 shows an example of the approach, on a 4-issue VLIW processor. The multiple SMMs operate independently, each connected to a specific lane of the processor, providing parallel access ports with a reduced cost. We extend [27] by taking into account the full-system operation, which provides important insights on the overall energy consumption. Additionally, this work implements SoMMA in the ρVEX processor [28] for more accurate performance evaluations, while Jost et al. [27] was applied to a monocycle VLIW processor. Further improvements from [27] include: a tree height reduction (THR) algorithm in the compiler framework; an additional *edge detection* benchmark; and data cache miss comparison between the modified and the baseline processors, showing further benefits of our architecture. Moreover, results in this paper can be partially found in the master's dissertation [29].

2.3 Contributions

The contributions of this work are as follows:

1. Reducing execution time and energy by avoiding unnecessary cache or Dynamic Random Access Memory (DRAM) accesses, since they are slower and more energy-consuming.
2. Allowing the usage of multiple software-managed memories in parallel and, consequently, providing a higher

```
for(i = 0; i < 7; i++){
    for(j = 0; j < 7; j++){
        var = 0;
        for(k = 0; k < 7; k++){
            var += smm_a[i][k] * smm_b[k][j];
        }
        res[i][j] = var;
    }
}
```

Fig. 4. Snippet of 7x7 matrix multiplication

memory throughput than other methods.

While previous works [16]–[18] also target (1), (2) is only provided by SoMMA in multi-issue processors. If one considers a multi-issue processor, like a VLIW, and the solutions presented in Fig. 3, SoMMA leverages the usage of multiple memories by simulating a multi-ported system with multiple single-ported memories. The figure also depicts how a vector structure could be potentially stored to benefit the multiple memories at once.

3 Example Application

This section presents an example application to demonstrate the application of SoMMA. Fig. 4 shows a snippet of C-code for a 7x7 matrix multiplication. Users can decide which variable is placed within the memories through a simple code annotation (variables beginning with *smm_*). We have chosen to place variables *a* and *b* (denoted as *smm_a* and *smm_b*) into the SMMs. That way, we can reason about the two different placements strategies for storing array variables in multiple memories.

We have compiled the application using with LLVM [12] using the –O3 flag, which enables many target-independent optimizations. Our algorithm works right before register allocation when machine instructions are within the target architecture and virtual registers are used. Fig. 5 shows a snippet of the assembly code for the application. The presented code loads values for variables *smm_a* (in BB#1) and *smm_b* (in BB#2) and multiplies values to compute a result per iteration (in #BB2).

Although programs are represented in a linear form at their last stage (assembly code), compilers can realize them as *data-dependence graphs* (DDG), where nodes represent instructions and edges represent the dependence between them [30]. An edge in the DDG is a relationship between a *definition* and its *use*. A *definition* represents an assignment of a variable, while a *use* represents its use as an operand. For instance, the *definition* of register %vreg23 (line 3) is within basic block BB#0, while its *use* is in BB#2 (line 16).

Our technique initiates by looking for *mov* instructions that use an SMM variable's address as operand. Such mov instructions are named as *def* instructions, because they define a starting point for an SMM variable. Two *def* instructions are observed within the code (lines 2 and 3), one for variable *smm_a* and another for *smm_b*. Registers used in *def* instructions are kept in a table of definitions and are used for searching memory instructions that are related to SMM variables. The algorithm will look for *uses* of such registers until it reaches a memory instruction. We say an instruction propagates an SMM variable if it reaches that variable through the definition-use relationship. For example, register *%vreg23* defines a starting point for an SMM variable (line 3). Its use is at line 16, which then defines another value, *%vreg24*, that is used at lines 17, 18 and 23, reaching three memory instructions for variable *smm_b*. The same idea is applied to variable *smm_a*. Memory instructions from lines 7 to 13 come from it. Through *definitions* and *uses*, the algorithm finds all memory instructions related to SMM variables, in order to replace them later.

Although it identifies instructions for SMM variables, the process above does not improve parallelism per se. The algorithm must also keep track of all offsets used for each variable. The more offsets within a basic block, the more memories can be used for the variable, and the more parallelism can be exploited. The number of memories should be proportional to the number of offsets encountered per basic block, as same instructions should be used in different iterations of a loop. For this example, the inner loop was unrolled seven times by LLVM optimizations. Thus, seven offsets are detected for variable *smm_a* in BB#1, and seven for *smm_b* in BB#2 (only three are displayed in Fig. 5). In a processor with eight software-managed memories, the compiler would infer the use of seven memories for each variable.

Moreover, we can also notice how the offset ranges differ between those variables. This disparity happens because, in one, *smm_a*, data is fetched in-row (distance between offsets is four because we are dealing with 4-byte integers), while in the other, *smm_b*, data is fetched in-column (distance is 28, seven values of 4-byte integer per row). Two different placements strategies should be adopted for maximizing parallelism, depending on the pattern of access. By identifying the stride in the accesses for each variable, the compiler chooses the placement for each array.

Fig. 6 shows the placement strategy used for the variables according to the observation of offsets in the basic blocks. For *smm_a*, values are spread through the SMMs in a contiguous way (one row at a time), while for smm_b, values are spread in a non-contiguous way (one column at a time), but within a known interval.

SMM 0	SMM 1	SMM 2	SMM 3	SMM 4	SMM 5	SMM 6	SMM 7
a[0,0]	a[0,1]	a[0,2]	a[0,3]	a[0,4]	a[0,5]	a[0,6]	
a[1,0]	a[1,1]	a[1,2]	a[1,3]	a[1,4]	a[1,5]	a[1,6]	
a[2,0]	a[2,1]	a[2,2]	a[2,3]	a[2,4]	a[2,5]	a[2,6]	
a[3,0]	a[3,1]	a[3,2]	a[3,3]	a[3,4]	a[3,5]	a[3,6]	
.	.	.	.	.	.	.	
.	.	.	.	.	.	.	
.	.	.	.	.	.	.	

SMM 0	SMM 1	SMM 2	SMM 3	SMM 4	SMM 5	SMM 6	SMM 7
b[0,0]	b[1,0]	b[2,0]	b[3,0]	b[4,0]	b[5,0]	b[6,0]	
b[0,1]	b[1,1]	b[2,1]	b[3,1]	b[4,1]	b[5,1]	b[6,1]	
b[0,2]	b[1,2]	b[2,2]	b[3,2]	b[4,2]	b[5,2]	b[6,2]	
b[0,3]	b[1,3]	b[2,3]	b[3,3]	b[4,3]	b[5,3]	b[6,3]	
.	.	.	.	.	.	.	
.	.	.	.	.	.	.	
.	.	.	.	.	.	.	

Fig. 5. Offset address locations for a 7x7 matrix multiplication. The example shows how matrix smm_a (shortened to a) and smm_b (shortened to b) would be placed in the SMMs. First scheme, on the left, shows an allocation through column order, while the second, on the right, displays an allocation through row order.

Finally, after gathering all information needed, the compiler generates code for the preambles and replaces memory instructions for SMM instructions, which will be described in next section. Instruction scheduling and bundling take place afterward. During these steps, the SMM units on each lane give greater parallelism opportunities for the heuristics. In this example, load instructions in BB#2 and BB#3 will execute in different units, thus, parallelizing memory access.

```
1   BB#0:
2           %vreg20 = MOVi <ga:@smm_a>;
3         %vreg23 = MOVi <ga:@smm_b>;
4   BB#1:
5           %vreg21 = ADDr %vreg19, %vreg18;
6           %vreg22 = ADDr %vreg20, %vreg21;
7           %vreg7 = LDW %vreg22, 20;
8           %vreg6 = LDW %vreg22, 16;
9           %vreg8 = LDW %vreg22, 24;
10          %vreg5 = LDW %vreg22, 12;
11          %vreg4 = LDW %vreg22, 8;
12          %vreg3 = LDW %vreg22, 4;
13          %vreg2 = LDW %vreg22, 0;
14  BB#2:
15          %vreg10 = PHI %vreg17, <BB#1>, %vreg11,
<BB#2>;
16          %vreg24 = ADDr %vreg23, %vreg10;
17          %vreg25 = LDW %vreg24, 140;
18          %vreg26 = LDW %vreg24, 168;
19          %vreg27 = MPYLUr %vreg26, %vreg7;
20          %vreg28 = MPYLUr %vreg25, %vreg6;
21          %vreg29 = MPYHSr %vreg26, %vreg7;
22          %vreg30 = MPYHSr %vreg25, %vreg6;
23          %vreg31 = LDW %vreg24, 112;
24          %vreg32 = MPYHSr %vreg31, %vreg5;
25          %vreg33 = MPYLUr %vreg31, %vreg5;
26          %vreg34 = ADDr %vreg28, %vreg30;
27          %vreg59 = CMPNEBRegi %vreg11, 0;
            .
            .
            .
50          STW %vreg58, %vreg9, 0;
```

Fig. 6. Snippet of assembly code.

4 SoMMA Architecture

Memory bandwidth is a cornerstone of high-performance processors. The memory must provide enough data such that functional units (FUs) are frequently occupied, as idle FUs also consume static energy while producing no useful values. A solution for bandwidth improvements relies on adding smaller and faster memory units, known as scratchpads, that can benefit from data reuse. Scratchpads provide better performance and energy efficiency than caches because no tag array and replacement logic are necessary. The control of these memories is done at software level, so they are typically known as software-managed memories (SMMs) or software-controlled memories. SMMs are particularly beneficial when reuse can be guaranteed, i.e., one needs to assure that temporal and spatial locality are present in the program.

Our main goal is to mitigate memory complexity in multi-issue processors by creating a memory architecture that explores the use of multiple memories in parallel. That way, single-ported memories combined with the data cache provide to the system a higher bandwidth without the overhead of adding new ports. We use a solution that incorporates changes in both hardware and software and can overcome the limit of single-port systems, providing a higher throughput.

4.1 Hardware

We propose a method that requires minimum hardware changes to support our SMMs. By limiting most transformations to software, we remove the costs of complex hardware structures to manage SMMs. The main difference between SMMs and caches relies on who controls them. The former are controlled by software and requires no extra hardware for tag storage and checking while the latter are hardware-controlled and the software has little control over where data is placed in memory.

Fig. 7 shows an example of SoMMA within a hypothetical 4-issue VLIW processor. A memory architecture is created with the placement of a software-managed memory within each parallel lane. Except Lane0, which can also access the regular data memory hierarchy through the L1 data cache (MEM block in Fig. 7), each lane can only access its own memory's content. Because lanes can only operate in one memory, there is no need to specify which memory SMM instructions should access. Each lane only sees its own memory, and the address spaces are independent of one another. Thereby, we avoid the costs of having multi-ported memories, while still providing parallelism for the processor. SMM instructions operate in the same way as loads and stores: one register as the base register, an immediate offset that will be added to the base register, and a second register that will hold the value that must be stored to or was loaded from memory.

```
1 for each BB in BFS Order do
2 begin
3
4    for each Instr in BB do
5    begin
6       if instr is Call or Branch
7          continue;
8
9       if isSMMVariableDef(Instr) then
10         add Instr to ListDefVar
11      else
12         if PropagatesSMMVariable(Instr) then
13            track register on Var
14
15            if InstrIsStoreOrLoad(Instr) then
16               EvaluateOffset(Instr)
17               add Instr on Var
18            end if
19         end if
20      end if
21   end for
22 end for
```

Fig. 7. High-level algorithm for Variable Propagation

4.2 Software

Section 4 used a matrix application to exemplify how our approach analyzes code and performs modifications. This section presents each step of the process, how they interact and their key features. Our algorithm is divided into two main procedures we call variable discovery and variable transformation. Variable discovery collects information about the variables for the software-managed memories while variable transformation makes the necessary code changes to use SMMs for each discovered variable in the previous process. We will discuss both procedures in the next subsections.

4.2.1 Variable discovery

This is the process responsible for finding out which variables can be placed in the SMMs. The user can specify which variables may be stored in the memories through code annotations.

Fig. 8 shows a simplified version of our algorithm for variable discovery. The algorithm runs in all functions of the program, searching for variables that are allowed to be on the SMMs. It starts by traversing the basic blocks in a breadth-first search (BFS) order (Line 1). The algorithm basically searches for def instructions and memory instructions according to the basic block order. It aims at finding instructions that define variables, and instructions that propagate the use of these variables through basic blocks (BBs). The core of the algorithm begins at line 4, where we iterate over all instructions in the basic block. Lines 6 and 7 show a quick escape for call and branch instructions since they do not need to be checked. The algorithm may execute four functions at lines 9, 12, 15 and 16 with the following purposes:

- isSMMVariable (Line 9): this function checks for def instructions within basic blocks, looking for the code annotation given by the user. If the annotation is found, the instruction is inserted in the list of def instructions for that variable (Line 10), keeping track of its defined register.
- PropagatesSMMVariable (Line 12): This function verifies if the current instruction propagates any SMM variable through the definition-use relationship, keeping track of definition registers. At line 16 of Fig. 5, e.g., %vreg23 is propagated, and a new definition is found (%vreg24).
- InstrIsStoreOrLoad (Line 15): if the current instruction is a memory instruction, the algorithm should evaluate the offset and insert the instructions into a list of memory instructions for further replacement.
- EvaluateOffset (Line 16): we aim at using software-managed memories primarily on vector and matrix variables inside loop statements since they offer more possibilities of parallelism. When a loop is unrolled, multiple iterations overlap and different offsets can be found in a single larger iteration. Offsets are important because they tell us exactly where the data is located in memory, and therefore, they will help us in the process of placing values properly on our memories. This function evaluates the offset for the current instruction. The compiler builds a table for offsets of variables for deciding on the number of SMMs necessary for each variable.

We construct a list of all variables and its memory-related instructions (Line 17), preparing all necessary data structures for the variable transformation process. This process also keeps track of the location for each memory instruction added to the variable list (to which basic block it belongs).

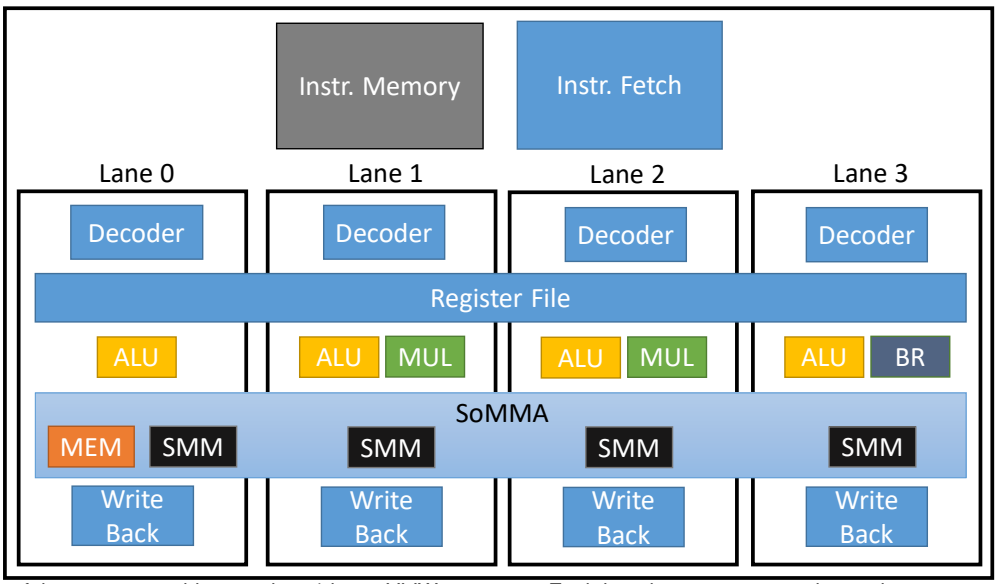

Fig. 8. Location of the memory architecture in a 4-issue VLIW processor. Each lane has access to an internal memory.

4.2.2 Variable Transformation

After collecting the required information about SMM variables, we still need to perform code transformations in order to use them. We should analyze the information gathered previously, and make decisions about whether code changes are necessary. Variable Transformation is where all the code changes take place. These changes go from code modification in memory instructions to new code insertion.

We analyze the variable and quantify how many software-managed memories are necessary depending on the number of offsets observed in the basic blocks. The number of memories allocated to a variable must be multiple of the number of offsets within a single basic block, always respecting the maximum number of SMMs available in the processor. For instance, a 7x7 matrix multiplication benchmark, shown in Fig. 5, unrolls the inner loop by a factor of 7 in the compiler framework. After analyzing all basic blocks, we conclude that at most seven different offsets can be spotted in a single basic block. Therefore, our algorithm will infer 7 SMMs for *smm_a* and *smm_b* (suppose that we have a total of eight), enabling our processor to fetch multiple data in parallel.

Moreover, when we cannot guarantee that data is accessed uniformly, our compiler detects that values cannot be spread out across multiple memories. A simple example would be calculating a sum of values in a vector that randomly selects a position. There is no way to assure where the value is located when using more than one SMM. In this case, our algorithm selects only one memory. We will call single-memory variables those that only use one SMM, and multi-memory variables the ones that use more than one SMM. The compiler is also aware of the processor's characteristics, such as number of software-managed memories, arithmetic and logic units (ALUs) and number of ports to the main memory hierarchy.

Some variables may require previous placement within the software-managed memories before their use. Others may not demand that task since their values are firstly calculated and then inserted in the memories. The preamble code is only inserted when the first memory instruction spotted in the program that is related to the variable is a load. Otherwise, no preamble code is necessary, as we are already filling up our memories as the program executes. Moreover, this process substitutes regular memory instructions, that is, those that use the standard memory hierarchy, for software-managed memory references. We also have to guarantee that a single instruction can be used for iterating over multiple reference locations, independently of the executed iteration.

4.2.2.1 Offset Calculation

Preamble insertion firstly needs to calculate the offset for each variable location. Single-memory variables are easily calculated, as only one memory is necessary. Multi-memory variables, on the other hand, require extra calculations, since different addresses of one variable can live in different SMMs. Our technique uniformly assigns locations to the values depending on the number of SMMs calculated previously. Two separate relocation strategies may occur:

- 1-consecutive allocation: the next data value fetched within the basic block will have a one-unit distance from the previous value, and thus, we fetch data in a contiguous manner. Variable smm_a in Fig. 5 uses this approach.
- N-consecutive allocation: this scheme occurs when two data addresses within the basic block are far from one another, regarding their memory locations. This scheme deals with data that are non-contiguous in memory. N corresponds to the distance between two data offsets within a same basic block. Variable smm_b uses this approach.

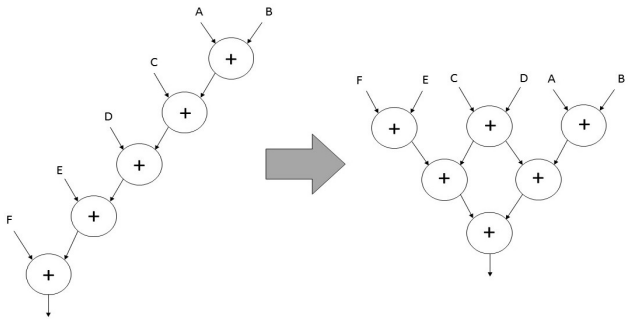

Fig. 9. Tree Height Reduction Algorithm

Fig. 6 illustrates both schemes in a practical example. Each column represents a software-managed memory, and each row accounts for a location within it. We use the same 7x7 matrix multiplication algorithm to clarify the difference between these two schemes. Our algorithm infers the use of 7 SMMs for each variable. The allocation scheme on the left can be employed for variable smm_a which is accessed with successive addresses, i.e., the 1-consecutive allocation is employed. Variable smm_b fetches data from different rows, and therefore, the N-consecutive allocation is applied (where N is equal to 7 for this case). The allocation scheme depends on the distance between offsets within a basic block.

4.2.2.2 Preamble Insertion and number of software-managed memories

For those variables that require preamble, we insert new basic blocks in the program, before variables are accessed. The preamble is the code segment responsible for fetching values that should reside in the SMMs from the regular memory hierarchy, i.e., from the main memory through the L1 data cache. The compiler inserts instructions to read these values and place them in the software-managed memories, using the proper offset according to the information collected in the previous process. Preamble insertion adds some execution time for each variable placed in the SMMs, for that reason, it is important to choose variables that present a great data reuse.

4.2.2.3 Instruction transformation

After the assignment of variables to locations, the final step is code transformation. Here the compiler checks for all definitions and memory references found in the variable discovery process and assigns new offsets and addresses for them. Our algorithm computes lanes and offsets according to the allocation schemes described in the previous subsection.

4.2.3 Additional optimizations

Some traditional compiler optimizations have important synergy with SoMMA. Namely, loop unrolling and tree height reduction provide important gains when coupled with it.

Loop unrolling transformation has proven to be very efficient to boosting performance on our software-managed memories. It allows optimizing the program through the execution of multiple loop iterations at the same time. Loop unrolling needs to guarantee that addresses do not alias. That way, iterations may overlap, and performance may increase through unrolling. This transformation helps improving performance by allowing multiple offsets at the same basic block. Therefore, more software-managed memories can be used to provide parallel accesses more easily.

Algorithms, such as matrix multiplication, x264, and the

sum of absolute differences (SAD), are characterized by having a long addition tree after the computation of each iteration. When the loop is unrolled, the result of each operation is summed with the previously computed values, in an inefficient way. This long chain of additions has little parallelism, limiting the performance gains of the SMA. Therefore, we have also implemented a tree height reduction algorithm [31], that efficiently computes and reduces the height of the addition tree for most of the algorithms.

Fig. 9 shows the code transformation performed by our compiler. This algorithm reduces the tree height from n to $\log_2 n$, and therefore, a significant performance increase can be observed as more parallelism is exposed. Bear in mind that both techniques were also applied to the baseline processor, as they also benefit processors with standard memory architectures.

4.3. Extending the technique to Superscalar Processors

With some ISA customization, it is possible to extend our technique to work not only with static multi-issue processors, like VLIWs, but also with superscalar processors. VLIW processors characterize for having multiple execution lanes, with each lane being capable of executing different sets of operations. Because lanes can only operate in only one memory, the same instruction opcode can be used for any SMM, i.e., the memory used depend only on the lane location.

Superscalars, on the other hand, are multi-issue processors that execute operations dynamically, according to resource availability. Using our technique in superscalar processors requires multiple instruction opcodes to differentiate the access of each SMM unit. For instance, a superscalar processor that dispatches eight instructions at a time can potentially be extended with eight SMMs, and therefore, requires eight new instructions. Multiple instructions guarantee that SMMs have different address spaces and operate independently, and, consequently, allowing the use of the technique in superscalar processors.

5 Experimental Setup and Results

SoMMA requires the use of a multi-issue processor for maximizing ILP, as the architecture demands multiple units operating at the same time. The LLVM compiler framework [12] was, thus, extended to support a VLIW processor and SoMMA was also added as part of the framework. The modularity of LLVM showed to be a perfect fit for our work, as new optimizations can easily be included.

5.1 VLIW Example (VEX) architecture

VEX is a 32-bit clustered VLIW architecture based on the Lx/ST200 [32], a family of VLIW processors used to power embedded media devices. In order to apply the software changes required by the proposed memory architecture, we have added support for the VEX ISA to the LLVM compiler toolchain as part of our work. All measurements were done considering the ρVEX processor [28].

5.2 Experimental Setup

Our tests were done using a modified version of the VEX Simulation System [33], which already provides a built-in level-one cache simulator. We have customized new instructions to handle SMMs and added internal memories to the tool to emulate the access to these memories. Processors use an 8-issue configuration. Those that implement our technique are denoted as SoMMA processors. Processors that make no use of software-managed memories are the baselines, denoted as regular processors.

Experiments consist of measuring speedup, energy consumption and the energy-delay product (EDP) on a full-system that includes processor, instruction and data caches, and DRAM memory. Results are shown for two scenarios that aim at maintaining similar area between processors. The same amount of memory added with SMMs was subtracted from the L1 data cache, thereby maintaining the same total memory. The first scenario consists of a baseline processor with 8 KB of data cache, and a modified processor with 4 KB of data cache, and eight 512 Bytes SMMs, for a total of 8 KB. In the second scenario, we extend the size of data storage to a total of 32 KB. Thus, the baseline processor has 32 KB of data cache, and the modified processor has 16 KB of data cache, and eight 2 KB SMMs. These are conservative approaches as the extra memory locations on the cache require additional tag storage. ICache of 8 KB and 32KB were used for each scenario, respectively, so that instructions and data memory have equivalent space. Furthermore, we consider systems with 2 GB of Low-power DDR2 (LPDDR2) memory, as we target embedded systems.

Cacti-p [6] was used to measure energy from cache and SMMs components, using a 65nm technology. The tool provides information on static power, and read and write energy cost per accesses, which allows us to estimate the static and dynamic energy consumption for each of these memories. DRAM energy consumption was measured using the DRAMPower [34], a DRAM power and energy estimation tool for a variety of standards, such as DDR2, DDR3, and LPDDR memories. For the processor core, we use energy consumption measurements from [35], [36], obtained through Cadence Encounter RTL compiler, using a 65 nm CMOS cell library from STMicroeletronics.

We have selected 9 benchmarks: a discrete Fourier transform (DFT), an edge detection algorithm (edge), a Fourier inverse transform algorithm (itver2), three matrix multiplication with different matrix sizes (m10x10, m16x16 and m32x32), the sum of absolute differences (SAD), the Knuth–Morris–Pratt string matching (kmp), and an x264 video encoder algorithm. These benchmarks provide data reuse and regular access patterns, which allow the use of our technique. The choice of using three matrix multiplication benchmarks comes from how our technique may change performance depending on the workload. LLVM generates different intermediate code for each benchmark, mainly due to different choices regarding loop unrolling.

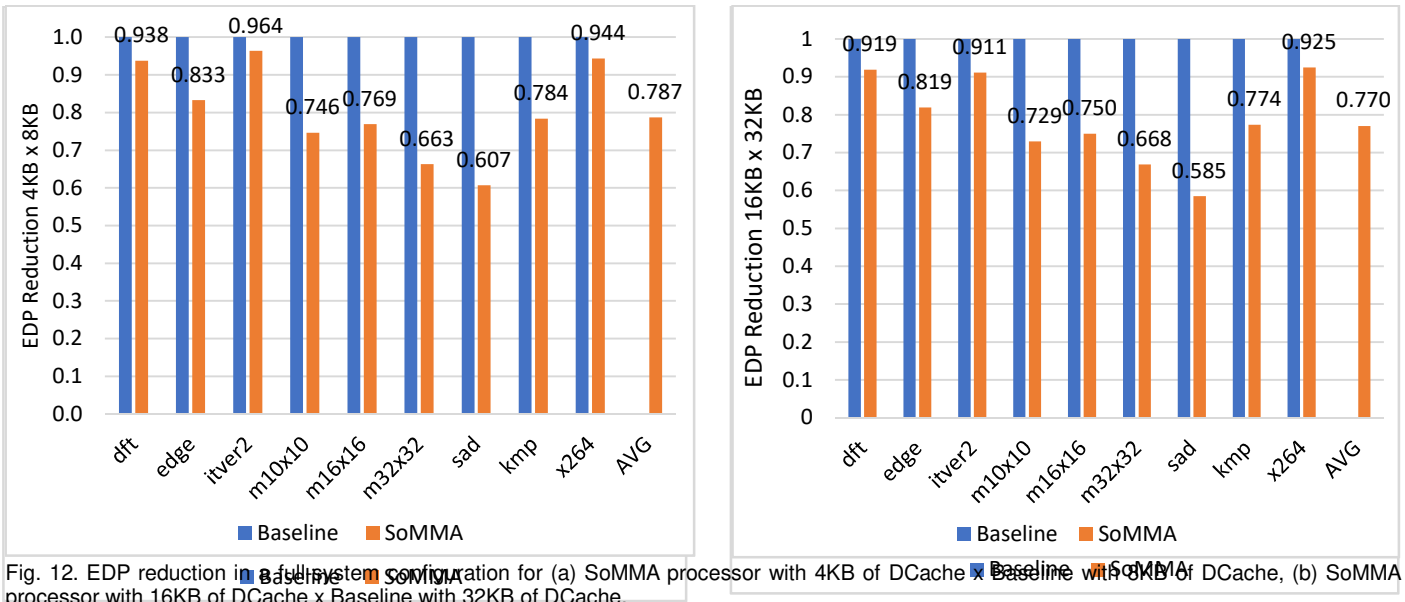

Fig. 12. EDP reduction in a full-system configuration for (a) SoMMA processor with 4KB of DCache x Baseline with 8KB of DCache, (b) SoMMA processor with 16KB of DCache x Baseline with 32KB of DCache.

Fig. 10. Normalized Speedup for (a) SoMMA processor with 4KB of DCache x Baseline with 8KB of DCache, (b) SoMMA processor with 16KB of DCache x Baseline with 32KB of DCache.

5.3 Results

5.3.1 Performance

Fig. 10 shows the speedup of the modified processors normalized by their respective baseline counterparts. Benchmarks *dft*, *x264* and *itver2* can provide good parallelism even for a single-port system, thus SoMMA was not capable of significantly improving performance for those benchmarks. We observe an average speedup of 1.118x and 1.121x in 4KB vs. 8KB and 16KB vs. 32 KB comparisons, respectively.

5.3.2 Energy

Full-system energy comparison between processors respects the following equation:

$$E_{total} = E_{proc.} + E_{ICache} + E_{DCache} + E_{DRAM}$$

where the energy is the sum of static and dynamic energy consumption.

Fig. 11 depicts the full-system energy reduction, showing the normalized energy consumption for each component of the system (DRAM, processor, ICache, and Data Memory). The Data memory (DMem) component consist of either DCache and SMMs (in SoMMA processors), or DCache (in baselines).

We have reduced energy consumption for almost every component in the applications. The insertion of the preamble code increased ICache accesses and misses in the SoMMA processor, though without having an impact on the overall energy consumption. Results show an energy reduction of 11% when comparing an SoMMA processor with 4KB of DCache (and 4KB of SMMs) and the baseline with 8KB of DCache, and 12.8% in the comparison of an SoMMA processor with 16KB of DCache (and 16KB of SMMs) and a baseline with 32KB of DCache. Moreover, we notice how most of the energy consumption stem from main memory and processors, as a consequence of performance improvements and, as covered in subsection 5.3.4, a reduction in the amount of accesses to the main memory.

5.3.3 Energy-delay Product (EDP)

We have also evaluated the energy-delay product (EDP) for the benchmarks. These results are the product of performance

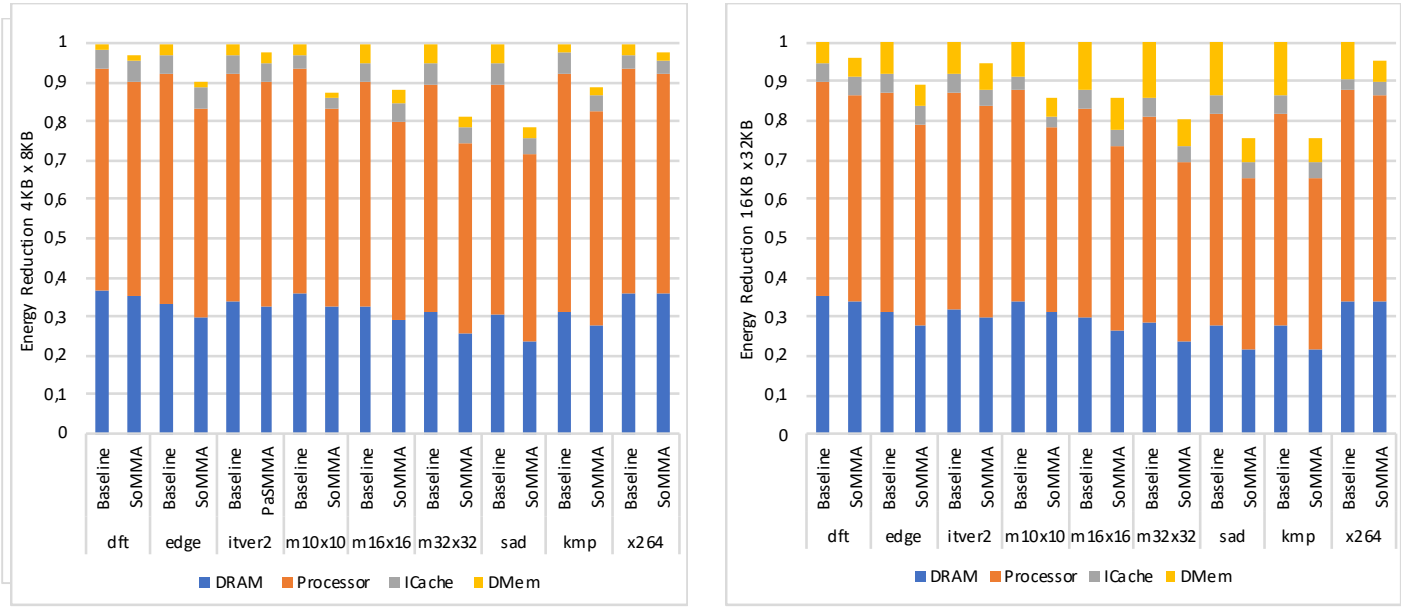

Fig. 11. Normalized Energy in a full-system configuration for (a) SoMMA processor with 4KB of DCache x Baseline with 8KB of DCache, (b) SoMMA processor with 16KB of DCache x Baseline with 32KB of DCache.

Fig. 13. Data cache miss reduction for (a) SoMMA processor with 4KB of DCache x Baseline with 8KB of DCache, (b) SoMMA processor with 16KB of DCache x Baseline with 32KB of DCache.

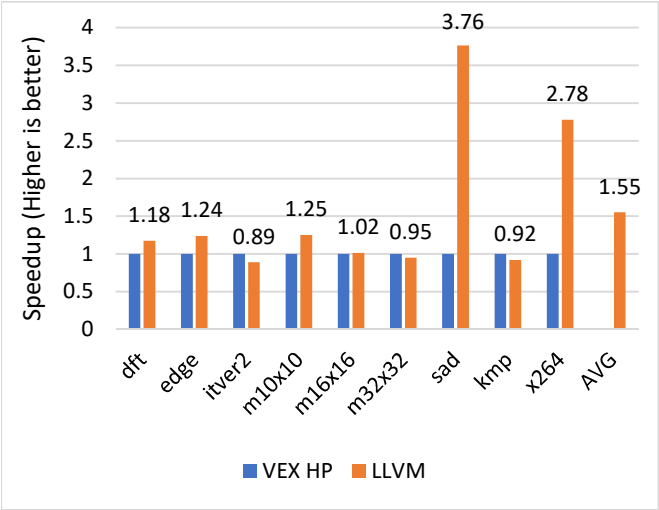

Fig. 14. Comparison between our LLVM code generator and the original VEX compiler

and energy reported in previous subsections. Fig. 12 presents the comparison between processors, showing that SoMMA is more effective when it comes to EDP, because we show better results for both performance and energy consumption. We have reduced EDP by a factor of 21.3% in our 4KB vs 8KB comparison, and 23% when comparing SoMMA processor with 16KB of DCache (and 16KB of SMMs) and a baseline with 32KB of DCache.

5.3.4 Data Cache Misses

Our last experiment consisted of comparing the number of data cache misses between SoMMA and baseline processors. Although we consider SoMMA processors with half the data cache size of baseline processors due to the presence of SMMs to maintain area equivalence, SoMMA leverages data reuse to reduce data cache misses in many of the benchmarks.

Fig. 13 illustrates the data cache miss comparison between processors. It is noticed how some benchmarks, namely *m10x10*, *m16x16*, *sad*, and *x264*, can accommodate all application data in cache, leading to the same number of data cache misses in all processors. In applications that have a larger working set have reduced data cache misses in SoMMA processors, as many of those requests are replaced by SMM accesses and there is no need to accommodate such data into the cache. In our 4KB vs. 8KB comparison, six out of nine benchmarks showed a reduction in cache misses. As caches get larger, the difference between SoMMA and baseline narrows to only three cases. These results show that our software-based managing of data can exploit locality that traditional caching algorithms miss, thereby reducing overall miss rates.

5.3.5 LLVM vs. original VEX Compiler

Because our approach provides a tight integration between software and hardware, with SMMs operating in synergy with the generated code, we were compelled to create a new code generator so that we have full control of how code is generated. Since this approach may raise concerns on how our implementation compares to the original compiler by Hewlett Packard (HP) for the VEX ISA, the remaining of this section provide details on this comparison.

The choice for using LLVM comes from its widely adoption in the research community and industry [37]. Its modular and reuse-centered design allows optimizations to run independent from the architecture where programs will execute. Therefore, this modularity helps backend designers to focus on target-specific optimizations, like VLIW packetizing, while the majority of compiler optimizations can run target-independently.

Fig. 14 shows the comparison between our LLVM-based code generator and the original VEX compiler for the same set of benchmarks considered previously. For these experiments, we make no use of the SMM memories as the intent is a comparison of benchmarks with equivalent resources and features. We notice that our LLVM compiler has comparable, and often better, performance over the baseline compiler. The main improvement comes from the powerful loop optimizations provided by LLVM. Though the baseline compiler also provides a series of loop optimizations, in two specific cases, it was not able to fully find the parallelism of these applications. If we discard the two singular speedups, i.e., *sad* and *x264*, our compiler is 6.3% better than the baseline. Considering all benchmarks, we observe an average speedup of 1.55x in comparison to the baseline.

Lastly, the goal of these experiments is not to demonstrate that the LLVM compiler has better overall performance than the original VEX HP compiler. We would only like to point out that for the set of applications considered in these experiments, our code generation seems to leverage the wide number of optimizations provided by LLVM in a more automatic fashion, resulting in a better performance. Perhaps the performance of the VEX compiler could be improved through profiling, as is one of the its main features. In fact, our tests focus mainly on the ability of the compiler to find optimizations automatically, and no profiling is used. One clear advantage of LLVM, however, is the support for recent versions of the C language, while the VEX compiler only supports the ISO C89 standard.

With this performance comparison in mind, we observe that, through the use of our technique, there is room for improvements in performance and energy consumption, and using our compiler solution has no negative impact on the executed applications, therefore, justifying its implementation.

6 Conclusion

This work proposed SoMMA, a software-managed memory architecture for embedded multi-issue processors that combines software-managed memories and the data cache. SoMMA aims at providing reductions in energy end EDP while still offering a better ILP through the use of multiple software-managed memories in parallel. We have developed an LLVM-based backend that analyzes and transforms code to operate such memories in cooperation with the memory hierarchy. Results have shown average speedups of 1.118x and 1.121x in a set of benchmarks for the ρVEX, when comparing two modified processors and their baselines. Energy consumption was reduced to 89% and 87.2%, while EDP showed reductions of 21.3% and 23%, all within experiments of full-system configurations with equivalent area.

Our algorithm currently works with global variables and plans on extending the work to handle stack variables, and heap-allocated variables are part of the future works. We also plan on extending the technique to work with other ILP-capable architectures, e.g., superscalar processors. As to better explore the uniqueness of each applications, we could vary the size of caches and scratchpads as to optimize performance for each application separately. For example, *kmp* has no inherently parallelism that makes use of all scratchpads. Two SMMs suffices to accelerate the application and reduce energy consumption. Moreover, new allocation schemes [38] could also be explored to support new applications. Future work is also envisioned within these contexts.

ACKNOWLEDGMENT

This work was supported in part by a grant from CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico, Brazil).

References

[1] G. E. Moore, "Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff.," *IEEE Solid-State Circuits Soc. Newsl.*, vol. 11, no. 5, pp. 33–35, 2006.

[2] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[3] J. Lee, H. Kim, and R. Vuduc, "When Prefetching Works, When It Doesn't, and Why," *ACM Trans. Arch. Code Optim.*, vol. 9, no. 1, pp. 2:1--2:29, Mar. 2012.

[4] W. A. Wulf and S. A. McKee, "Hitting the Memory Wall: Implications of the Obvious," *SIGARCH Comput. Arch. News*, vol. 23, no. 1, pp. 20–24, Mar. 1995.

[5] Y. Tatsumi and H. J. Mattausch, "Fast quadratic increase of multiport-storage-cell area with port number," *Electron. Lett.*, vol. 35, no. 25, pp. 2185–2187, 1999.

[6] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "CACTI-P: Architecture-level Modeling for SRAM-based Structures with Advanced Leakage Reduction Techniques," in *Proceedings of the International Conference on Computer-Aided Design*, 2011, pp. 694–701.

[7] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted MOSFET's with very small physical dimensions," *IEEE J. Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, Oct. 1974.

[8] M. Verma, L. Wehmeyer, and P. Marwedel, "Cache-aware scratchpad allocation algorithm," in *Proceedings of the conference on Design, automation and test in Europe-Volume 2*, 2004, p. 21264.

[9] S. Steinke, L. Wehmeyer, B.-S. Lee, and P. Marwedel, "Assigning program and data objects to scratchpad for energy reduction," in *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings*, 2002, pp. 409–415.

[10] F. Angiolini, F. Menichelli, A. Ferrero, L. Benini, and M. Olivieri, "A Post-compiler Approach to Scratchpad Mapping of Code," in *Proceedings of the 2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 2004, pp. 259–267.

[11] O. Avissar, R. Barua, and D. Stewart, "An optimal memory allocation scheme for scratch-pad-based embedded systems," *ACM Trans. Embed. Comput. Syst.*, vol. 1, no. 1, pp. 6–26, Nov. 2002.

[12] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, 2004, pp. 75--.

[13] J. A. Fisher, P. Faraboschi, and C. Young, *Embedded computing: a VLIW approach to architecture, compilers and tools*. Elsevier, 2005.

[14] M. Moazeni, A. Bui, and M. Sarrafzadeh, "A memory optimization technique for software-managed scratchpad memory in GPUs," in *Application Specific Processors, 2009. SASP '09. IEEE 7th Symposium on*, 2009, pp. 43–49.

[15] J. Pfrimmer, K. F. Li, and D. Rakhmatov, "Balancing scratchpad and cache in embedded systems for power and speed performance," in *IEEE-NEWCAS Conference, 2005. The 3rd International*, 2005, pp. 107–110.

[16] M. Verma and P. Marwedel, "Overlay techniques for scratchpad memories in low power embedded processors," *Very Large Scale Integr. Syst. IEEE Trans.*, vol. 14, no. 8, pp. 802–815, 2006.

[17] S. Udayakumaran, A. Dominguez, and R. Barua, "Dynamic allocation for scratch-pad memory using compile-time decisions," *ACM Trans. Embed. Comput. Syst.*, vol. 5, no. 2, pp. 472–511, 2006.

[18] M. Horro, G. Rodriguez, J. Tourino, and M. T. Kandemir, "Architectural exploration of heterogeneous memory systems," *arXiv Prepr. arXiv1810.12573*, 2018.

[19] V. Venkataramani, M. C. Chan, and T. Mitra, "Scratchpad-Memory Management for Multi-Threaded Applications on Many-Core Architectures," *ACM Trans. Embed. Comput. Syst.*, vol. 18, no. 1, p. 10, 2019.

[20] A. Shrivastava, N. Dutt, J. Cai, M. Shoushtari, B. Donyanavard, and H. Tajik, "Automatic management of software programmable memories in many-core architectures," *IET Comput. Digit. Tech.*, vol. 10, no. 6, pp. 288–298, 2016.

[21] S. Li, Y. Wei, and L. Ju, "Automatic data placement for CPU-FPGA heterogeneous multiprocessor System-on-Chips," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2019, pp. 1379–1384.

[22] B. Flachs *et al.*, "The microarchitecture of the synergistic processor for a cell processor," *IEEE J. Solid-State Circuits*, vol. 41, no. 1, pp. 63–70, Jan. 2006.

[23] W. Che and K. S. Chatha, "Scheduling of stream programs onto SPM enhanced processors with code overlay," in *Embedded Systems for Real-Time Multimedia (ESTIMedia), 2011 9th IEEE Symposium on*, 2011, pp. 9–18.

[24] G. A. Malazgirt, H. E. Yantir, A. Yurdakul, and S. Niar, "Application specific multi-port memory customization in FPGAs," in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, 2014, pp. 1–4.

[25] H. Bajwa and X. Chen, "Low-Power High-Performance and Dynamically Configured Multi-Port Cache Memory Architecture," in *Electrical Engineering, 2007. ICEE '07. International Conference on*, 2007, pp. 1–6.

[26] C. E. LaForest and J. G. Steffan, "Efficient multi-ported memories for FPGAs," in *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, 2010, pp. 41–50.

[27] T. Jost, G. Nazar, and L. Carro, "An Energy-Efficient Memory Hierarchy for Multi-Issue Processors," in *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2017.

[28] S. Wong, T. Van As, and G. Brown, "ρ-VEX: A reconfigurable and extensible softcore VLIW processor," in *ICECE Technology, 2008. FPT 2008. International Conference on*, 2008, pp. 369–372.

[29] T. T. Jost, "SoMMA: a software managed memory architecture for multi-issue processors," Programa de Pós-Graduação em Computação, Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2017.

[30] K. Cooper and L. Torczon, *Engineering a compiler*. Elsevier, 2011.

[31] D. L. Kuck, *Structure of Computers and Computations*. New York, NY, USA: John Wiley & Sons, Inc., 1978.

[32] P. Faraboschi, G. Brown, J. A. Fisher, G. Desoli, and F. Homewood, "Lx: A Technology Platform for Customizable VLIW Embedded Processing," *SIGARCH Comput. Arch. News*, vol. 28, no. 2, pp. 203–213, 2000.

[33] Hewlett-Packard Laboratories, "VEX Toolchain." [Online]. Available: http://www.hpl.hp.com/downloads/vex/.

[34] K. Chandrasekar *et al.*, "DRAMPower: Open-source DRAM Power & Energy Estimation Tool." [Online]. Available: http://www.drampower.info.

[35] A. L. Sartor, A. F. Lorenzon, L. Carro, F. Kastensmidt, S. Wong, and A. C. S. Beck, "Exploiting Idle Hardware to Provide Low Overhead Fault Tolerance for VLIW Processors," *J. Emerg. Technol. Comput. Syst.*, vol. 13, no. 2, pp. 13:1--13:21, 2017.

[36] A. L. Sartor, S. Wong, and A. C. S. Beck, "Adaptive ILP control to increase fault tolerance for VLIW processors," in *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2016, pp. 9–16.

[37] The LLVM Compiler Infrastructure, "Publications." [Online]. Available: http://llvm.org/pubs/.

[38] T. Jost, G. Nazar, and L. Carro, "Improving Performance in VLIW Soft-core Processors through Software-controlled ScratchPads," in *Embedded Computer Systems: Architectures, Modeling, and Simulation*

(SAMOS), 2016 International Conference on, 2016.

Tiago Trevisan Jost received the BSc. in computer engineering and MSc. In computer science from the Federal University of Rio Grande do Sul (UFRGS), Porto Alegre, Brazil, in 2014 and 2017.
He is currently a PhD candidate at the Commissariat à l'énergie atomique et aux énergies alternatives (CEA), Grenoble, France. His research interests include compiler optimizations and transformations for embedded systems, approximate computing, numerical analysis and scientifc computing applications.

Gabriel Luca Nazar received the B.Sc. degree in computer engineering from Universidade Federal do Rio Grande do Sul (UFRGS), Porto Alegre, Brazil, in 2010 and the Ph.D. degree in computer science from this same institution in 2013. He held a postdoctoral position at UFRGS from 2013 to 2014.
Since 2014 he is a professor at the Department of Applied Informatics, at the Institute of Infomatics of UFRGS. His research interests include embedded systems, reconfigurable systems, computer architecture and fault tolerance.

Luigi Carro (M'97) was born in Porto Alegre, Brazil, in 1962. He received the B.Sc. degree in electrical engineering, and the M.Sc. and Ph.D. degrees in computer science from the Universidade Federal do Rio Grande do Sul (UFRGS), Porto Alegre, in 1985, 1989, and 1996, respectively. He was with the Research and Development Group, STMicroelectronics, Agrate, Italy, from 1989 to 1991.
He is currently a Full Professor with the Department of Applied Informatics, Informatics Institute, UFRGS, and in charge of computer architecture and organization courses at the undergraduate levels. He is also a member of the Graduation Program in Computer Science at UFRGS, where he is co-responsible for courses on embedded systems, digital signal processing, and VLSI design. His current research interests include embedded systems design, validation, automation and test, fault tolerance for future technologies, and rapid system prototyping.