An Energy-Efficient Memory Hierarchy for Multi-Issue Processors

Tiago Jost, Gabriel Nazar, Luigi Carro
Instituto de Informática
Universidade Federal do Rio Grande do Sul
Porto Alegre, Brazil
{ttjost, glnazar, carro}@inf.ufrgs.br

***Abstract*—Embedded processors must rely on the efficient use of instruction-level parallelism to answer the performance and energy needs of modern applications. However, a limiting factor to better use available resources inside the processor concerns memory bandwidth. Adding extra ports to allow for more data accesses drastically increases costs and energy. In this paper, we present a novel memory architecture system for embedded multi-issue processors that can overcome the limited memory bandwidth without adding extra ports to the system. We combine the use of software-managed memories (SMM) with the data cache to provide a system with a higher throughput without increasing the number of ports. Compiler-automated code transformations minimize the effort of programmers to benefit from the proposed architecture. Our experimental results show an average speedup of 1.17x, while consuming 69% less dynamic energy and on average 74.7% lower energy-delay product regarding data memory in comparison to a baseline processor.**

Keywords—very long instruction word processor; bandwidth; memory architecture; software-managed memory; compiler

I. Introduction

Memory bandwidth has long been a limiting factor in performance for many applications. Even when exploiting access locality through cache memories to minimize this limitation, multi-ported memories are known to introduce significant costs. The study in [1] shows that the impact of increasing the number of ports in a memory system leads a quadratic increase for cell area. This result demonstrates why memory units are usually limited to very few ports for most systems. Our measurements with CACTI-P [2] corroborate those results. Figure 1 shows the normalized area for 32 KB cache memories with 1, 2, 8, and 16 ports for 22 nm and 65 nm technologies. We can observe, e.g., that 8 and 16-port caches at 22 nm showed area increases of more than 15x and 75x, respectively, when compared to a same-sized single-ported cache. Moreover, applications that struggle with memory bandwidth have their parallelism limited by the number of ports on the cache, as illustrated in Figure 2, where we show the potential speedup when one uses extra cache ports. The high area cost for such memories makes them unfeasible for today's embedded systems. Furthermore, the increase on the number of ports would also implicate on a higher energy consumption.

The complexity of caches, which include comparators and tag arrays, adds an extra layer of energy dissipation in the system. Applications recurrently use a memory access pattern that could be understood by software. This process would alleviate the use of comparators and tag arrays, and as a consequence, save energy on the access of the cache. This concept has already been explored in software-managed memories, trying to reduce cache power dissipation, like in [3], [4].

In this paper, we present a novel memory architecture system for embedded multi-issue processors. Its main goal is to obtain the bandwidth of a large multi-port cache with the cost and energy of a single port cache. We use a compiler-based approach to manage the extra memories and accelerate the execution of applications. Some previous works like [3] and [5] proposed to use software-managed memories, also called software-controlled memories or scratchpads, as replacement for caches aiming at reducing energy consumption; others [6], [7] focus on boosting performance of embedded processors due to the reduction of cache misses. A major difference between our strategy and scratchpads from previous works is how address spaces are treated. Scratchpads 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 software-managed 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. Thereby, the total available bandwidth can be increased, creating new opportunities for instruction-level parallelism (ILP) exploitation.

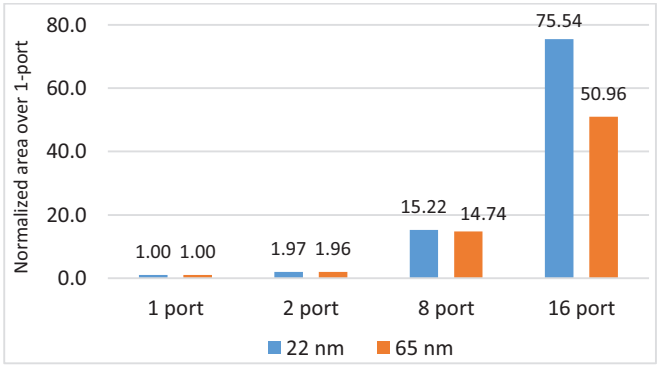

Fig. 1. Cache Area for different number of ports. Values were normalized over single port.

This work is sponsored by CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico, Brazil).

978-3-9815370-8-6/17/$31.00 ©2017 IEEE

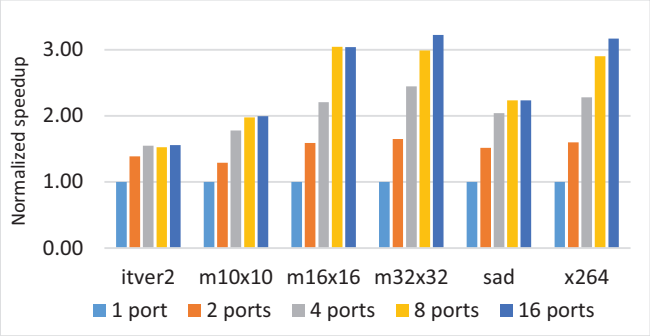

Fig. 2. Application speedup normalized over single-ported memory.

We demonstrate the efficiency of our technique in a very long instruction word (VLIW) processor, although our technique could be adapted to other ILP-capable architectures, e.g., superescalar processors. Experimental results show significant performance improvements while maintaining the same area of the standard memory hierarchy. Moreover, we analyze the energy efficiency for our memory system, showing significant energy and energy-delay product (EDP) reductions for the data memory hierarchy in comparison to a regular processor.

The rest of the paper is organized as follows. Section II presents related work. Section III gives a thorough explanation on the proposed technique. Section IV presents the experimental results, and section V concludes the paper.

II. Related Work

Many related works involve the use of software-managed memories as a replacement for caches in embedded processors [3], [6], [7]. Due to a simpler design, with no tag array and comparators, scratchpads provide considerable area and latency improvements over caches. There is no need for a hardware coherency mechanism, as software is responsible for their control. The two main allocation schemes are the static and overlay-based approaches. In the static-based approach, data is loaded once at the beginning of the program and remain invariant during its entire execution [6], [7]. On the contrary, data is loaded dynamically during the execution of the program in overlay-based allocation [4], [8].

The work in [3] shows an average reduction of 20-44% in energy consumption on instruction scratchpads, while Avisar et al. [8] use an automatic compiler method for statically allocating program data to scratchpad memories. In contrast to previous works [6], [7] that primarily focused on performance improvements due to decreasing miss penalty on the memory hierarchy, our technique achieves performance improvements through the exploitation of instruction-level parallelism (ILP) by parallelizing the use of the memories. The application executes parallel accesses through multiple memories. This way, the system's bandwidth is improved, leading to performance improvements. Our technique uses an overlay-based approach that allocates data into these memories on a function-basis.

Scratchpads are also being used in the industry. The work in [9] presents a memory optimization scheme that uses scratchpad memories for the NVIDIA G80 [10] graphics processing unit (GPU). The paper states that scratchpad memories help alleviating the pressure on global memory bandwidth, and therefore, maximize performance on data-intensive applications that have high usage of shared memory.

Other works [11]–[13] propose techniques to design multi-ported systems more efficiently. Some work focus on field-programmable gate array (FPGA) devices, where memory units, called block RAMs (BRAM), are typically dual-ported. Malazgirt et al. [12] presents an application-specific methodology that analyzes a sequential code, extracts parallelism and determines the number of read and writes ports necessary for such application. The work in [11] uses a more generic approach to create multi-ported memories in FPGAs. It efficiently combines BRAMs while achieving significant performance improvements over other methods. The work in [13] is applicable to application-specific integrated circuit (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. The work in [14] uses scratchpads to improve ILP and accelerate applications where data reuse is encountered through manually adding code to handle the software-managed memories. Our work is, therefore, an extension of [14].

III. Memory System Architecture

Memory bandwidth is the cornerstone of high-performance processors. The memory system should provide enough data such that functional units (FUs) are frequently occupied, as idle FUs also consume static energy while producing no useful values. The simplest solution to improve memory bandwidth comes from adding memory ports to the memory system. However, as presented in Figure 1, this solution has a deep impact in area, that cannot usually be paid in embedded systems. An alternative method for bandwidth increase relies on adding smaller and faster memory units that can benefit from data reuse, software-managed 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 combine with the data cache to provide the system with 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 high throughput. The rest of this section details hardware and software requirements in order to integrate the memory architecture in a system.

A. 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 is controlled by software and requires no extra hardware for tag storage and checking while the latter is

hardware-controlled and the programmer has little control over where data is placed in memory.

Figure 3 shows an example of how SMMs are placed within a hypothetical 4-issue VLIW processor. A memory architecture is created with the placement of a software-managed memory within each parallel lane. With the exception of Lane0, which can also access the regular data memory hierarchy through the L1 data cache (MEM block in Figure 3), 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.

B. Software

Hardware support is the first step to apply our technique to a processor. The second is the code generation process, where instructions that manipulate those memories are generated. Users can decide which variables are placed within the memories through simple code annotations. The code generation process detailed herein plays the main role in our approach.

We have modified the LLVM [15] compiler to support our software-managed memories. Our algorithm works right before register allocation, when machine instructions are in the target architecture and virtual registers are used. 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 [16]. 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. In Listing 1, for instance, the definition of *%vreg23* (line 3) is within basic block *BB#0*, while its use is in *BB#2* (line 16).

Our technique initiates by looking for *definitions* of global variables marked with the code annotation (variables *smm_a* and *smm_b* in lines 2 and 3). Registers used in these *definitions* are kept in a table and are used for searching memory instructions that are related to SMM variables. The algorithm will look for *uses* of those *definitions* until it reaches memory instructions. 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.* Through definitions and uses, the algorithm finds all memory instructions related to SMM variables, in order to replace them later.

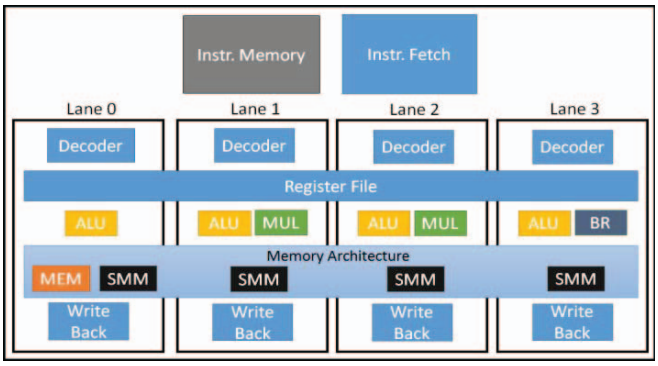

Fig. 3. Location of the memory architecture in a 4-issue VLIW processor. Each lane has access to an internal memory

1) Code Generation Process

Our algorithm is divided into two main actions 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 in case a variable is discovered in the previous process. We will discuss both actions in the next subsections, showing specific characteristics for each action and how they correlate.

a) 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. Our algorithm currently works only with global variables and plans on extending the work to handle stack variables and heap-allocated variables are part of the future works.

Listing 2 shows a high-level simplified version of our algorithm for *variable discovery*. The algorithm runs in all functions of the program, trying to find 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). It aims at finding instructions that define variables, and instructions that propagate the use of these variables through basic blocks

```
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;
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;
27          %vreg59 = CMPNEBRegi %vreg11, 0;
            .
            .
50          STW %vreg58, %vreg9, 0;
```

Listing 1. Snippet of assembly code.

(BBs). The core of the algorithm begins at line 4, where we iterate over all instructions in the basic block. Line 6 and 7 show a fast 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 Listing 1, 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 offset 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).

```
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
```

Listing 2. High-level algorithm for Variable Propagation

b) 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 collected 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.

Some variables may require previous placement within the software-managed memories before their use. Others may not require 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 must guarantee that a single instruction can be used for iterating over multiple reference locations, independently of the executed iteration.

1. Offset Calculation

Preamble insertion firstly needs to calculate the offset for each variable location. Variables that reside on a single memory are easily calculated. 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 different 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.
- 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.

If one considers a matrix multiplication algorithm as example (multiplying values of *a* times *b*), variable *a* would adopt 1-consecutive allocation, because data is accessed with successive addresses, while variable *b* fetches data from different rows, and therefore, the N-consecutive allocation is applied.

2. Preamble Insertion

For those variables that require *preamble*, we insert new basic blocks in the program, before variables are accessed. The compiler inserts instructions to read from main memory and place values of the variable inside the software-managed memories, using the proper offset according to the information collected in the previous process. Keep in mind that the preamble insertion will add some execution time for each variable, for that reason, the more data reuse in the variable, the better.

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 and base addresses are known when preamble insertion is performed.

IV. Experimental Results

In order to minimize the programmer's effort, the LLVM compiler framework [15] was extended to support SMMs. We have developed an LLVM backend for the VLIW Example (VEX) instruction set architecture (ISA) [17], developed by HP. VEX is also target as the architecture for the ρ-VEX processor [18], a reconfigurable processor that is implemented on top of an FPGA. A handful of tools [19], such as, customizable architecture and cache simulators, is provided, giving a complete simulation platform for programmers. In order to simulate, we have added new instructions to handle SMMs in the simulator.

We have conducted experiments to see how effective our technique is in terms of execution time, dynamic energy consumption in data memory hierarchy, the energy-delay product (EDP) in data memory hierarchy, as well. Results were obtained using 8-issue processors, with all instructions having a 1-cycle latency. Processors that implement our technique are denoted as SMMA processors, while those that make no use of software-managed memories are the regular processors.

Eight benchmarks were used to measure gains: a discrete Fourier transform (DFT), a Fourier inverse transform algorithm (itver2), three matrix multiplication with different matrices sizes (m10x10, m16x16 and m32x32), the sum of absolute differences (SAD), the KMP string matching, and an x264 video encoder algorithm. These benchmarks provide data reuse and regular access patterns, which allow the use of our technique. In order to maintain similar area, we have reduced the size of our level-one data cache on the SMMA processors proportional to the size of all SMMs together, that way processors have the same number of data memory locations. SMMs are 2 KBytes of size each, for a total of 16 KBytes. Thus, SMMA processors have 16 KBytes of data cache, while regular processors have 32 KBytes. This is a conservative approach as the extra 16 KBytes of cache of the baseline still require additional tag storage. As a result, measurements with CACTI-P [2] showed that our SMMA processor has a data memory area that is 7.6% smaller than the regular processor. Table I shows the individual energy costs per access obtained with CACTI-P and used to obtain the results in this section, considering a 65 nm technology. We have included energy results for SMMA with a 32 KB data cache as well. These results allow evaluating the gains exclusively from the introduction of the SMMs, as will be discussed in section IV.B.

TABLE I. Energy cost per access

	Read (pJ)	Write (pJ)
Cache 16 KB	24.51	26.05
Cache 32 KB	41.97	38.10
SMM 2 KB	3.80	7.05

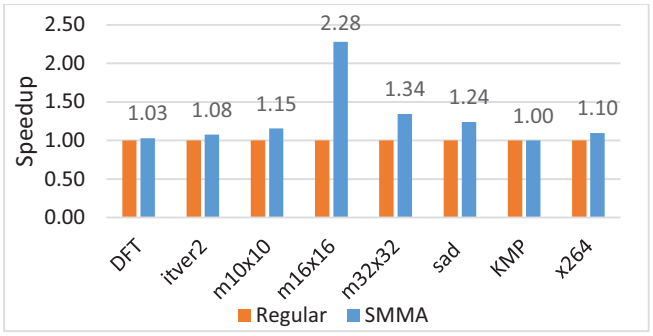

Fig. 4. Performance comparison normalized for the baseline

A. *Performance*

Figure 4 shows the performance comparison between our modified processor and the baseline. We have chosen to use three matrix multiplication benchmarks in order to show 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. For this reason, different performance improvements were observed. Results show performance improvements in 7 out of 8 benchmarks, with KMP as an exception, showing nearly the same performance as the baseline. SMMA results in Figure 4 are for 16 KB, as they were very similar to 32 KB. Our technique has shown a performance improvement of 1.17x on average.

B. *Energy*

Figure 5 illustrates the comparison between processors in regards of dynamic energy in the data memory hierarchy. Energy gains stem from two main reasons: each SMM access consumes substantially less than L1 accesses; the L1 size reduction allows L1 accesses to consume less as well. Figure 6 shows that, even though the total number of accesses was increased, most accesses to the L1 were replaced by SMM accesses. The SMMA 16 KB processor consumes 69% less energy in the data memory system compared to the baseline. Even in the KMP algorithm that showed no improvements in performance, 75% less dynamic energy was consumed when compared to the baseline. This happened because the application exhibits little ILP, but high locality, which could be

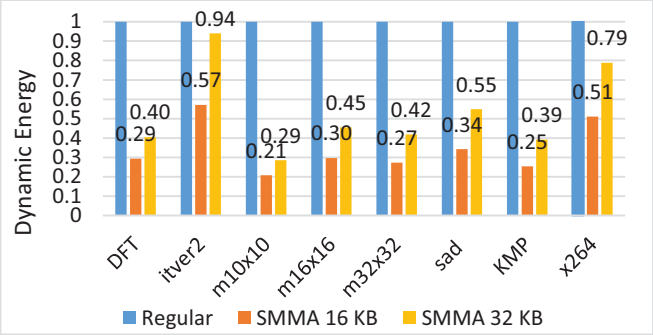

Fig. 5. Normalized dynamic energy.

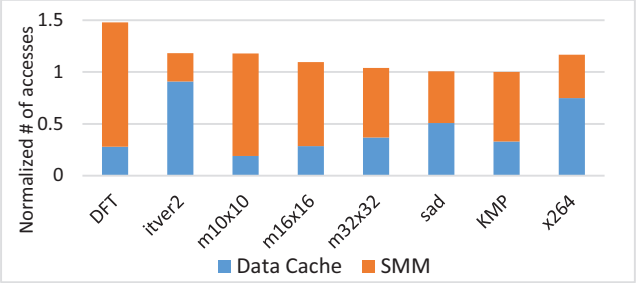

Fig. 6. Normalized number of accesses for the SMMA processor in comparison to the baseline

exploited by replacing most L1 accesses with SMM accesses. For *itver2* and *x264* gains were reduced since a smaller fraction of accesses could be diverted to SMMs.

When considering a 32 KB data cache, energy gains are reduced, since L1 accesses consume the same as the baseline. However, as SMMs allow us to make fewer L1 accesses, even in this scenario relevant energy gains were achieved (51.2 % on average).

C. Energy-delay product

We have also evaluated the energy-delay product for the benchmarks. These results are the product of the figures reported in the previous subsections. Figure 7 presents the comparison between processors, showing that our approach is effective when it comes to EDP. We have reduced EDP by a factor of 74.7% on average for the SMMA 16 KB, and 60.1% on average for the SMMA 32 KB.

V. Conclusion

This work proposed a memory architecture system for embedded multi-issue processors that combines software-managed memories and the data cache. 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 an average speedup of 1.17x in a set of benchmarks, while consuming 69% less dynamic energy and on average 74.7% lower energy-delay product regarding data memory in comparison to a baseline processor.

Future work will focus on extending the algorithm to handle stack and heap variables. The SMA is not restricted to VLIW processors, although special care will have to be taken when using this technique on other ILP-capable architectures, e.g., superscalar processors. Extending the technique for such case is a relevant future work.

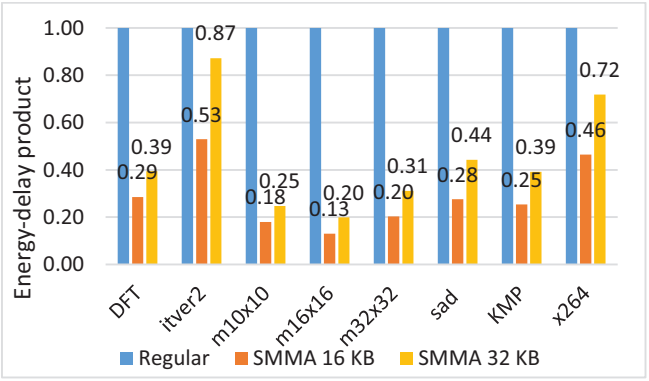

Fig. 7. Normalized EDP for applications.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] D. Kirk, "NVIDIA Cuda Software and Gpu Parallel Computing Architecture," in *Proceedings of the 6th International Symposium on Memory Management*, 2007, pp. 103–104.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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, p. 75--.

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

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

[18] 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.

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