A Transparent and Energy Aware Reconfigurable Multiprocessor Platform for Simultaneous ILP and TLP Exploitation

Mateus Beck Rutzig
Universidade Federal de Santa Maria
Departamento de Eletrônica e Computação
Santa Maria, Brasil
mateus@inf.ufsm.br

Antonio Carlos S. Beck, Luigi Carro
Universidade Federal do Rio Grande do Sul
Instituto de Informática
Porto Alegre, Brasil
{caco, carro}@inf.ufrgs.br

Abstract - As the number of embedded applications increases, companies are launching new platforms within short periods of time to efficiently execute software with the lowest possible energy consumption. However, for each new platform deployment, new tool chains, with additional libraries, debuggers and compilers must come along, breaking binary compatibility. This strategy implies in high hardware and software redesign costs. In this scenario, we propose the exploitation of Custom Reconfigurable Arrays for Multiprocessor Systems (CReAMS). CReAMS is composed of multiple adaptive reconfigurable processors that simultaneously exploit Instruction and Thread Level Parallelism. It works in a transparent fashion, so binary compatibility is maintained, with no need to change the software development process or environment. We also show that CReAMS delivers higher performance per watt in comparison to a 4-issue Superscalar processor, when the same power budget is considered for both designs.

keywords- reconfigurable system, multiprocessor, embedded systems

I. INTRODUCTION

Embedded systems are getting increasingly heterogeneous in terms of software. While a current high-end smartphone has a considerable number of applications running on a single platform, it must also provide the best possible energy/performance efficiency. To support such demands, embedded platforms (e.g. Texas Instruments OMAP, NVIDIA Tegra, Qualcomm Snapdragon, etc) comprise up to four simple general purpose processors (e.g. ARM) that are surrounded by several dedicated hardware accelerators (i.e. for communication, graphics and audio processors), each one of them implementing its particular instruction set architecture (ISA). While nowadays one can find up to 21 accelerators in an embedded platform, it is expected that 600 different dedicated acceleration engines will be needed to efficiently cover the whole set of application behaviors that will exist in the year 2024 [1].

Therefore, the current scenario for embedded platform development is not favorable for future embedded products. Heterogeneous processors/accelerators using different ISAs require specialized designers and tools, which have a negative effect on software productivity and time-to-market. Every release of a new platform will not be transparent to the software developers, since together with a new platform, a new version of its tool chain with particular libraries and compilers must be provided. Besides the obvious deleterious effects on software productivity and compatibility for any new hardware upgrade, there will be also intrinsic costs of new hardware developments for every new product.

On the other hand, multiprocessing systems that are conceived as the replication of the same processor make software development easier. As a unique ISA is used, software developers can generate code using the very same tool chain for the whole system in any platform version, which maintains full binary compatibility and ensures short time to market. However, a homogeneous multiprocessing platform does not provide an aggressive trade-off between energy and performance.

In this scenario, dynamic reconfigurable architectures appear as an alternative. They have already shown to be very attractive for embedded platforms, since they can provide on-the-fly heterogeneity by adapting the Instruction Level Parallelism (ILP) exploitation to the application requirements. However, they must also:

- exploit TLP, in order to efficiently attack the whole spectrum of application behaviors: those that contain dominant thread level parallelism and those single threaded applications with some ILP;
- be transparent, with no extra costs to the software development process, and to present satisfactory tradeoffs in terms of energy, performance and area;
- be composed of generic processing elements that could adapt to the particularities of a given application, emulating the behavior of the dedicated accelerators that have been successfully employed in current heterogeneous platforms;
- implement the same ISA, so binary compatibility can be sustained and software productivity would not be sacrificed.

In this work we show how to achieve the aforementioned objectives, by proposing a platform based on Custom Reconfigurable Arrays for Multiprocessor Systems. With CReAMS, it is possible to virtually emulate the desired heterogeneous behavior with a physical homogeneous platform, by using a binary translation mechanism that provides the demanded ILP and TLP for each application. As each processor can adapt its ILP on the fly, it can emulate the concept “the right processor for right task” of big-LITTLE ARM’s strategy [4], with the advantage of implementing regular circuit that is easier to manufacture in CMOS technologies. Different from traditional physical heterogeneous architectures, CReAMS presents the advantage that only one ISA is implemented. To the best of the authors´ knowledge, CReAMS is the only platform that provides the software productivity of a homogeneous multiprocessor device, with full software compatibility,

978-3-9815370-0-0/DATE13/©2013 EDAA

while sustaining the energy benefits of a heterogeneous organization.

This paper is organized as follows. Section 2 shows a review of researches regarding multi-threaded reconfigurable architectures. Section 3 presents the proposed architecture, followed by the methodology used to obtain results and the comparison of CReAMS with multiprocessing systems composed of single-issued processors and out-of-order superscalar processors, in Section 4. Finally, Section 5 concludes this work.

II. RELATED WORK

One can find different single- and multi- processing environments which applies some kind of adaptability to improve the performance of applications [11][12][13]. They can be homogeneous or heterogeneous, considering their architecture (i.e. what ISA is implemented) and organization (i.e. if the processors that comprise the system are the same or not).

Watkins [2] presents a procedure for mapping functions in the ReMAPP system, which is composed of a pair of coarse grained reconfigurable arrays that is shared among several cores. As an example of a system with homogeneous architecture and heterogeneous organization, one can find Thread Warping [3]. It extends the Warp Processing [10] system to support multiple-thread execution. In this case, one processor is totally dedicated to execute the operating system tasks needed to synchronize threads and to schedule their kernels in the accelerators. ARM's big-LITTLE [4] also implements a heterogeneous organization and homogeneous architecture by grouping a Cortex A7 and a Cortex A15 within the same chip. This strategy aims to provide high performance as well as power efficiency by selecting at run-time the right processor to execute the task at hand according to its requirements.

KAHRISMA [5] is another example of a totally heterogeneous architecture. It supports multiple instruction sets (RISC, 2- 4- and 6-issue VLIW, and EPIC) and fine- and coarse-grained reconfigurable arrays. Software compilation, ISA partitioning, custom instructions selection and thread scheduling are made by a design time tool that decides, for each part of the application code, which assembly code will be generated, considering its dominant type of parallelism and resources availability. A run time system is responsible to code binding and for avoiding execution collisions in the available resources.

Both ReMAPP and KAHRISMA are able to optimize multiple threads, but they break the binary compatibility.

In this work, we propose Custom Reconfigurable Arrays for Multiprocessor System that:

- Unlike KAHRISMA, Thread Warping and big-LITTLE ARM's strategy, our proposal is physically homogeneous in both architecture and organization. Heterogeneity is achieved on the fly, without any human intervention, by employing a binary translation mechanism that will be explained later. It eases the software development process since a well known tool chain (i.e. gcc) is used for any of its versions. Neither source code modifications nor additional libraries are necessary if new processing elements are inserted.
- KAHRISMA, Thread Warping and ReMAPP rely in special and particular tool chains to extract thread-level parallelism and to prepare the platform for execution. Our approach does not change the current development flow, so well-known application programming interfaces (e.g. OpenMP) can be used. This way, the programmer can extract TLP in a friendly interface, since such APIs are already coupled to a great number of compilers (e.g. gcc and icc), which makes the software development and the binary generation process easier than the aforementioned approaches.
- In contrast to ReMAPP and Thread Warping, CReAMS employs a coarse-grained reconfigurable fabric instead of a fine-grained one. Fine-grained architectures provide higher acceleration levels, but its scope is narrowed to applications that have few kernels responsible for a large part of the execution time. Coarse-grained reconfigurable architectures reduce the reconfiguration time and memory footprint due to the small context size, which increases its field of applications, because they are capable of accelerating the entire application.
- In contrast to ARM´s big.LITTLE strategy, we do not waste energy rediscovering parallelism like a superscalar does, but rather redefine the data path on-the-fly for ILP exploitation with minimum energy dissipation.

III. THE PROPOSED APPROACH

A general overview of the proposed platform is given in Figure 1(a). The thread level parallelism is explored by replicating the number of Dynamic Adaptive Processors (DAPs) (in the example of the Figure 1(a), by eight DAPs). In this way, we extend the single-thread based reconfigurable architecture presented in [6] to handle multithreaded applications. A DAP is a transparent coarse grained reconfigurable architecture coupled to the processor, and will be explained in details later. The communication among DAPs is done through a 2D-mesh Network on Chip using a XY routing strategy. CReAMS also includes an on-chip unified 512 KB 8-way set associative L2 shared memory, illustrated as SM in the Figure 1(a). We divided DAP in four blocks to better explain it, as illustrated in Figure 1(a). These blocks are discussed in the following sections.

Processor Pipeline (Block 2)

A SparcV8-based architecture is used as the baseline processor to work together with the reconfigurable system. Its five stage pipeline reflects a traditional RISC design (instruction fetch, decode, execution, data fetch and write

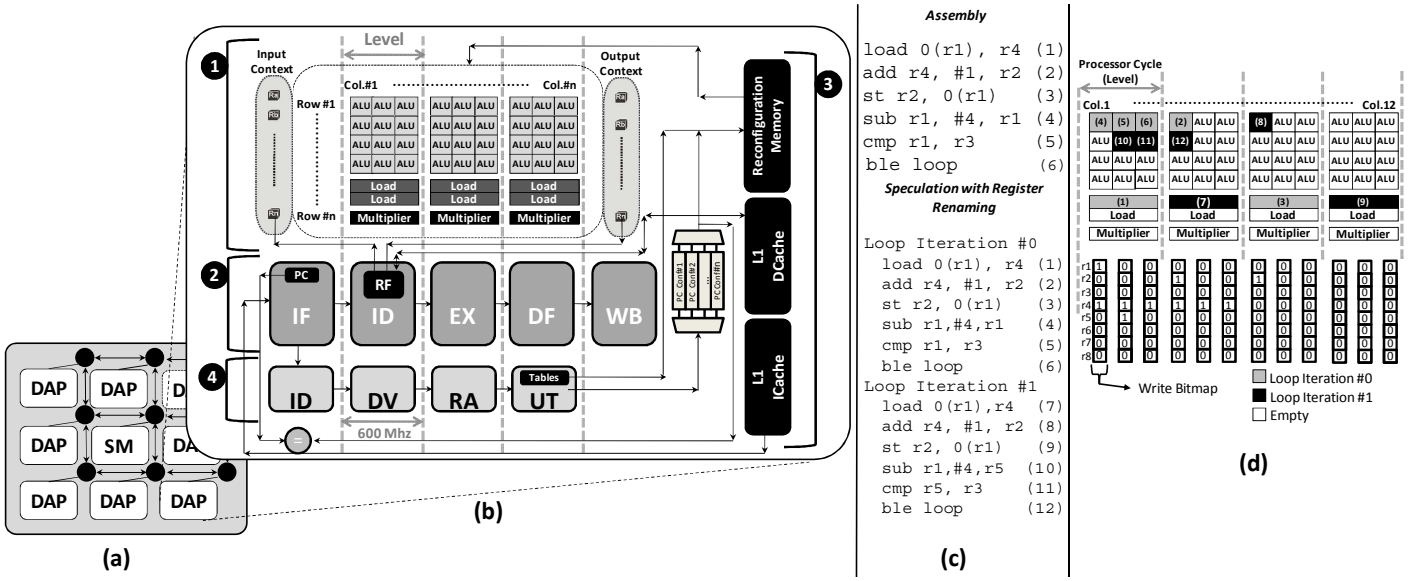

Figure 1. (a) CReAMS (b) DAP blocks (c) Assembly of a loop (d) Allocation inside of the reconfigurable data path

back) and it similar to other RISC processors used in well-known embedded platforms (e.g. MIPS, ARM).

Reconfigurable Data Path Structure (Block 1)

The reconfigurable data path is coarse-grained and tightly coupled to the SparcV8 pipeline, avoiding external accesses to the memory, saving power and reducing the reconfiguration time. As illustrated in the Figure 1(a), the data path is organized as a matrix of rows and columns. The number of rows dictates the maximum instruction level parallelism that can be exploited, since instructions located in the same column are executed in parallel. For example, the illustrated data path (Block 1 of Figure 1(a)) is able to execute up to four arithmetic and logic operations, two memory accesses (two memory ports are available in the L1 data cache) and one multiplication considering ideal assumptions (no true dependences between instructions). The number of columns determines the maximum number of data dependent instructions in sequence that can be executed in one configuration.

Three columns of arithmetic and logic units (ALU) compose a level. A level does not affect the SparcV8 critical path (which, in this case, is given by the multiplier circuit). Therefore, up to three data dependent ALU instructions can be executed in the reconfigurable data path within one SparcV8 cycle, without affecting its original frequency (600 MHz). Memory accesses and multiplications take one equivalent SparcV8 cycle to perform their operations. We have coupled sleep transistors [7] to each functional unit in the reconfigurable data path so it is possible to switch their power off when they are not used.

The entire structure of the reconfigurable data path is totally combinational: there is no temporal barrier among the functional units. The only exception is for the entry and exit points. The entry points are used to keep the input context and the exit points are used to store the results. Both structures are connected to the processor register file.

The feeding of the input context with the necessary data is the first step to configure the data path before starting execution. Results are sent to the SparcV8 register file on demand. It means that if any value is produced at any data path level (a cycle of SparcV8 processor) and if it will not be changed in the subsequent levels, this value is written back in the cycle after that it was produced. If there are more writes in a given column than the number of available ports in the register file, they are forwarded to the next level. In the current implementation, the SparcV8 register file has two write/read ports.

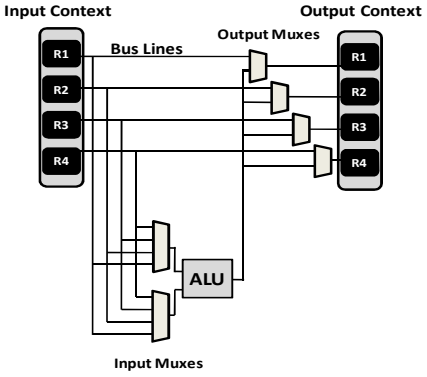

Figure 2. Interconnection mechanism

A simplified overview of the interconnection structure of the reconfigurable data path is shown in Figure 2. Each input register is connected to the functional units by bus lines. Multiplexers are responsible for choosing the correct value. Each functional unit has two multiplexers (input multiplexers) in their inputs: they make the selection of the issuing operands from the input context. There is a multiplexer for each bus line (output multiplexers) that will choose which result from the set of functional units of the column will be bypassed to the next column, and so on.

The size of the input and the output contexts may be a limit to the number of instructions allocated in a single data path configuration. When the input context is full, a new configuration will be created. Therefore, a small input context size may degrade performance because a new configuration may be created, even if there still available functional units. On the other hand, an excessively large input context would lead to a huge overhead in the data path area, since extra output multiplexers and larger input multiplexers would be necessary.

Storage Components (Block 3)

The address cache holds the memory address of the first instruction of every configuration built by the dynamic detection hardware. It is used to verify the existence of a configuration in the reconfiguration memory (where configuration bits are kept): an address cache hit indicates that a configuration was found. The address cache is implemented as a 4-way set associative table containing 64 entries.

Besides the two storage components explained before, the current DAP implementation has a private 32 KB 4-way set associative L1 data cache and a private 8 KB 4-way set associative L1 instruction cache. We have reduced the size of the instruction cache (from 32KB of the original SparcV8 standalone processor to 8KB) because, according to our experiments, the same hit rate is achieved by the SparcV8 in the DAP compared to the standalone SparcV8 using a quarter size of its 32KB L1 instruction cache. This happens because, as translated instructions are stored in the reconfiguration memory, the SparcV8 within the DAP has fewer memory accesses to the L1 instruction cache than the standalone SparcV8 processor. Thus, while the impact of the additional area of the address cache and the reconfiguration memory in the DAP design is amortized, the performance results have not been biased to our favor.

Dynamic Detection Hardware (DDH) (Block 4)

The hardware responsible for instruction detection and allocation in the data path, named as Dynamic Detection Hardware (DDH), is implemented as a 4-stage pipelined circuit. It does not increase the critical path of the SparcV8 processor. It is a binary translation mechanism that translates instructions from SparcV8 ISA to data path configurations. The stages of the circuit are divided in Instruction Decode (ID), Dependence Verification (DP), Resource Allocation (RA), Update Tables (UT). The translation process is performed as the GPP executes the instruction, so there is no extra performance overhead.

For each data path´s column of figure 1(a) there is a bitmap responsible for storing the target operands of the already allocated instructions in the respective column, named as Write Bitmap (Figure 1(d)). Thus, the source operands of each incoming instruction are compared to the target operands in this bitmap to verify in which column the current instruction should be allocated, according to their data dependencies.

Figure 1(c) shows the assembly code of a loop. In the right side of the same Figure the allocation of this code sequence inside the reconfigurable data path (Figure 1(d)) is demonstrated. The first incoming instruction, a memory access operation, is allocated at the highest functional unit of the leftmost data path column, considering the type of this operation. In this process, as this type of operation takes an entire level (which is an equivalent processor cycle) and covers three data path columns, the fourth bit of the write bitmap of the columns 1, 2 and 3 should be set to maintain the allocation consistency.

The dependence detection starts from the second instruction. In our example, the instruction number two reads the R4 register. As it is written by the previous instruction, a read after write (RAW) dependence is found. The DDH detects it and allocates the instruction number two at the later column of instruction number one. The second bit of the fourth write bitmap is set since this instruction has the register R2 as the target operand. The dependence analysis follows these steps until instruction number five.

As the DDH supports speculation, when the branch (instruction number six) comes to the DDH pipeline, it sets the speculation flag and the next incoming instructions will still be allocated in the same configuration. It means that the iteration number zero and iteration number one of Figure 1(c) are placed in the same data path configuration.

The instruction number seven (the first instruction of iteration one) has a data dependence with the instructions number one and number two. But it can be executed in parallel with the instruction number two: while it will read the register with data dependence in column three, instruction number seven will only write it in the column six. As can be seen in the final allocation (Figure 1(d)), DDH dynamically performs software pipelining by overlapping instructions of different loop iterations.

In the allocation of instruction number ten, the DDH detects that R1 (written by instruction number four) could be read by the incoming instruction in the second column but could not write in this register at this column. Thus, it performs register renaming by allocating R1 to the next empty register of input context. All subsequent instructions that contain a reference to R1 are modified accordingly.

Some additional features handled by the DDH:

- Memory accesses are allowed. It is conservative when considering load/store allocation: when a store comes to the DDH, all subsequent loads are allocated after it.
- As already explained, it handles false dependences (WAW and WAR, so multiple pointers to the same register reference are allowed. Therefore, multiple enters of the same reference to a register can be allocated in the context bus.
- It performs speculative execution. In this case, each operand that will be written back has an additional flag

indicating its depth concerning its level of speculation (i.e. in which basic block it originally was). When the branch relative to that basic block is taken, it triggers the writes of these correspondent operands. The speculative policy is based on bimodal branch predictor.

IV. RESULTS

We have compared CReAMS to two different systems: a multiprocessing platform, built by the replication of standalone single-issue 5-stage pipelined SparcV8 processors, named MPSparcV8; and a multiprocessing system composed of 4-issue out-of-order SparcV8 processors. For both comparisons, an on-chip unified 512 KB 8-way set associative L2 shared memory is implemented in both multiprocessing systems. They work at the same operating frequency of the original processor (600MHz) and communicate through a 2D-Mesh NoC infrastructure using the XY routing strategy that also works at the same frequency of the processors. All circuits are implemented in VHDL and synthesized to CMOS 90nm technology using Synopsys Design Compiler.

Benchmarks from different suites were selected to cover a wide range of different behaviors in terms of type (i.e. TLP and ILP) and degree of existing parallelism. From the OpenMP Source Code Repository, Splash2 and Parsec, we have selected *md*, *jacobi* and *lu* that are, due to their nature, applications in which TLP is dominant. Three applications (*apsi*, *equake* and *ammp*) were chosen from SPEC OMPM2001, and four applications (*susan edges*, *susan smoothing*, *susan corners* and *patricia*) from the MiBench suite, which reflects a traditional embedded scenario. They were single–threaded applications parallelized by hand using POSIX and OpenMP libraries to take advantage of multiprocessing environments.

Each DAP has a reconfigurable data path (Block 1 of the Figure 1(a)) composed of 6 arithmetic and logic, 4 load/store and 2 multipliers units per column. The entire reconfigurable data path has 24 columns. A 46 KB reconfiguration memory, implemented as a DRAM memory, is able to store up to 64 configurations. The configurations are indexed by a 64-entries 4-way set associative Address Cache.

CReAMS *versus Single-issued SparcV8 multiprocessor system (MPSparcV8*

One can consider that most portable devices are battery dependent, with severe power constraints. Therefore, we have also evaluated the results considering a peak power budget of 3 Watts for both architectures, which is the foreseen limit value for batteries in the coming years [1]. The peak power of the standalone SparcV8 is 385.14 mWatts, while the DAP consumes 699.33 mWatts. By consequence, we have compared the 8-SparcV8 MPSparcV8 against the CReAMS composed of 4-DAP.

Table 1 shows the execution time, in seconds, considering the whole benchmark set. As can be seen, considering the same power budget scenario, CReAMS outperforms the MPSparcV8 execution time in seven benchmarks (*equake*, *apsi*, *ammp*, *susan_edges*, *patricia*, *jacobi* and *lu*), even considering applications that contain massive TLP, such as *lu* and *susan edges*. *ammp* is the application that benefits the most from the dynamic nature of CReAMS when the same peak power budget is considered, achieving 43% shorter execution time. On average, CReAMS reduces the execution time by 15%.

Table 1. Performance, Energy and Energy-Delay considering a power budget

	8SparcV8	4DAPs	8SparcV8	4DAPs	8SparcV8	4DAPs
	Time (in ms)		Energy (in Joules)		Energy-Delay	
equake	1349	1113	226.0	151.9	305	169
apsi	8149	7111	1275	894	10386	6361
ammp	12794	7197	1732	758	22155	5452
susan_e	174.4	138.6	40.98	24.94	7.15	3.46
patricia	116.2	88.3	16.85	16.67	1.96	1.47
susan_c	44.70	44.77	18.20	11.53	0.8134	0.5160
susan_s	410.3	416.8	249.7	100.0	102.5	41.7
md	0.519	0.359	0.329	0.185	0.000171	0.000066
jacobi	178.6	249.5	115.7	69.6	20.7	17.4
lu	0.470	0.363	0.194	0.113	0.000091	0.000041

Energy savings are achieved because our architecture consumes less power, since besides executing sequences of instruction more efficiently in combinational logic, there are fewer memory accesses for instructions in the L1 instruction cache, even if one considers the extra power which is spent because of the DDH and reconfigurable data path. Once instructions are translated to a data path configuration, they will reside in the reconfiguration memory, which will avoid extra cache accesses in the future. In addition, energy savings are also obtained in a single access to the L1 instruction cache, since it is four times smaller in CReAMS than in the MPSparcV8. Furthermore, in several cases (e.g. with loops), the array needs to be reconfigured only once for several iterations, which also avoids several accesses to the reconfiguration memory.

We also measured the Energy Delay Product EDP for both platforms. CReAMS achieves better EDP in all benchmarks, being on average 47% lower.

CReAMS *versus Out-Of-Order Superscalar SparcV8 multiprocessor system*

So far, we have compared CReAMS against single-issue SparcV8 multiprocessing systems. Now, we compare CReAMS with a processor with aggressive ILP exploitation.

As there is no hardware implementation of a 4-issue OOO SparcV8 processor publicly available, we have modeled some of its characteristics (e.g. the size of the reservation stations and reorder buffers) based on the organization of a MIPS R10000 processor, since both SparcV8 and MIPS implement a very similar ISA. We have also used the same

MIPS R10000 processor to gather data about power consumption [9].

For comparison purposes, we have created two different scenarios that consider a power budget of 27 Watts and 53 Watts. As one DAP consumes 699.33 mWatts, the power consumption of four OOO SparcV8 superscalar processors and an architecture composed of 32DAPs are similar [9]. Therefore, both fit in the first scenario. For the second scenario, we have compared eight OOO SparcV8 superscalar processors with our platform composed of 64DAPs.

For these experiments, we have chosen a subset of the applications to better illustrate the versatility of our proposal. This subset is composed of the most representative applications considering TLP and ILP availability. As already discussed, *blackscholes*, *swaptions* and *jacobi* have perfect load balancing, so they are very suitable for TLP exploitation and have tiny room for ILP exploitation. *susan corners* is quite the opposite: it presents a lot of ILP, but it is significantly unbalanced. Finally, *lu* represent applications in which neither ILP nor TLP are available.

Table 2 shows the execution time of both systems considering the two scenarios of power budget. CReAMS outperforms the 4-issue OOO MPSparcV8 in all TLP-oriented applications in both scenarios. 32-DAP is 4.51, 3.26 and 3.68 times faster than 4-core OOO MPSparcV8 when running *swaptions*, *blackscholes* and *jacobi*, respectively. When the power budget increases to 53 Watts (second scenario), the gains of 64-DAPs over 8-Core 4-issue OOO MPSparcV8 remain in the same proportion.

CReAMS shows performance improvements of 28% and 8.5% when the power budgets #1 and #2 are considered, which shows that the proposed approach delivers higher performance per watt than a multiprocessing system based on a 4-issue OOO superscalar processors.

V. CONCLUSIONS

In this work, we have proposed the idea of custom reconfigurable arrays for multiprocessing systems, aiming to offer the advantages of a heterogeneous multiprocessor architecture and to improve software productivity, since no modifications in the tool chain or code recompilations are necessary for its use. Considering designs with the same chip area or with a constrained peak power budget, our proposal offers considerable improvements in performance and energy consumption for a wide range of different software behaviors, compared to a traditional multiprocessor environment. In addition, the efficiency of the adaptability provided by CReAMS is demonstrated in a comparison to state-of-the-art processor, a multiprocessing system composed of 4-issue Out-of-Order SparcV8 processors.

Table 2. Execution time of 4-issue OOO MPSparcV8 and the proposed approach

	4OOOSparcV8	*8OOOSparcV8*	*32DAPs*	*64DAPs*
		Execution Time (ms)		
susan_c	16.448	11.221	12.838	10.344
swaptions	7.094	3.551	1.572	0.792
blackscholes	3.408	1.710	1.048	0.535
jacobi	123.841	62.245	33.665	19.233
lu	0.352	0.265	0.279	0.310

REFERENCES

[1]International Technology Roadmap for Semiconductors: Available at http://www.itrs.net/links/2009ITRS/2009Chapters_2009Tables/2009_SysDrivers.pdf

[2]Watkins, M.A.; Albonesi, D.H., "Enabling Parallelization via a Reconfigurable Chip Multiprocessor", Workshop on Parallel Execution of Sequential Programs on Multi-core Architectures, 2010. 37th International Symposium on Computer Architecture, June 2010.

[3] Lee, J., Wu, H., Ravichandran, M., and Clark, N. 2010. Thread tailor: dynamically weaving threads together for efficient, adaptive parallel applications. ISCA '10. pp. 270-279.

[4] Available at http://www.arm.com/products/processors/technologies/bigLITTLEprocessing.php

[5] Koenig, R.; Bauer, L.; Stripf, T.; Shafique, M.; Ahmed, W.; Becker, J.; Henkel, J.; "KAHRISMA: A Novel Hypermorphic Reconfigurable-Instruction-SetMulti-grained-Array Architecture," Design, Automation & Test in Europe Conference, pp.819-824, 2010

[6] Beck, A.C.S, Rutzig,M.B., Gaydadjiev,G., and Carro,L.. Transparent reconfigurable acceleration for heterogeneous embedded applications. In Proceedings of the conference on Design, automation and test in Europe (DATE '08). ACM, New York, NY, USA, 1208-1213.

[7] Shi, K.; Howard, D. "Challenges in Sleep Transistor Design and Implementation in Low-Power Designs". In Proceedings of Design Automation Conference, 43. 2006, pp. 113 – 116.

[8] Kavvadias, S. A Trace Driven Configurable SparcV8 ISA Simulator, 2001. Available at: <http://www.ics.forth.gr/~kavadias/SMT_Page/trace_driven_configurable_simulator.htm>.

[9]Heinrich, J. MIPS R10000 User Manual. MIPS R10000 User Manual, 1997. Available at <http://techpubs.sgi.com/library/manuals/2000/007-2490-001/pdf/007-2490-001.pdf>.

[10] Roman Lysecky, Greg Stitt, and Frank Vahid. 2004. Warp Processors. In Proceedings of the 41st annual Design Automation Conference (DAC '04). ACM, New York, NY, USA, 659-681.

[11] Beck, A.C.S and Carro, L. *Dynamic Reconfigurable Architectures and Transparent Optimization Techniques*, Springer-Verlag, 2010.

[12] Beck, A.C.S., Lisboa, C.A. and Carro, L. *Adaptable Embedded Systems*, Springer-Verlag, 2012.

[13] Beck, A.C.S, Rutzig,M.B., Gaydadjiev,G., and Carro,L..Run-Time Adaptable Architectures for Heterogeneous Behavior Embedded Systems. In Proceedings of the 4th international workshop on Reconfigurable Computing: Architectures, Tools and Applications (ARC '08)