doi:10.1093/comjnl/bxh000

Local Memory Design Space Exploration for High Performance Computing

Ramon Bertran*[1], Marc Gonzàlez[2], Xavier Martorell[1], Nacho Navarro[2] and Eduard Ayguadé[1]

[1] *Barcelona Supercomputing Center Cr. Jordi Girona 29, 08034 Barcelona, Spain.*
[2] *Department of Computer Architecture, Universitat Politècnica de Catalunya, Cr. Jordi Girona 1-3, 08034 Barcelona, Spain.*
* *Corresponding author: rbertran@ac.upc.edu*
Email: {rbertran,marc,xavim,nacho,eduard}@ac.upc.edu

The performance of High Performance Computing (HPC) applications highly depends on the memory subsystem due to the huge data sets used that do not fit into the cache hierarchy. Besides, energy efficiency has become a main design factor and, consequently, both performance and energy efficiency are primary goals in HPC designs. As a result, energy efficient high performance memory subsystems designs should be explored. In this paper, we extend the architecture of general-purpose processors by adding a software managed local memory (LM) and a very simple programmable DMA controller (PDC). We demonstrate that with these extensions –together with an efficient run-time management– we improve performance and energy consumption factors. We perform an LM design space exploration study for a Intel® Pentium® 4 platform: we analyze performance, energy and energy-delay product (EDP) for a total of 27 computational loops of the NAS benchmarks. We show a 1.2x performance speedup factor and an energy reduction of 6.21% on average when using a constrained 32KB LM with commodity memory bandwidths (6.4GB/s). More aggressive configurations (i.e., 256KB LM + 12.8GB/s) show at least 2.14x performance speedup factors and energy savings of 42.07% on average.

Keywords: hybrid memory hierarchy, software managed cache, design space exploration, performance and energy evaluation

Received 00 Month 200X; revised 00 Month 200X; accepted 00 Month 200X

1. INTRODUCTION

The momentum behind chip multiprocessors (CMPs) has led these architectures to be the choice for current and future processors in a wide spectrum of application domains such as high performance computing (HPC), commodity desktops, gaming or embedded systems. Most of the current CMP designs are based on the replication of several cores –onto the same chip– which are connected to the memory subsystem, and it is the memory subsystem design which is particularly crucial given the relationship between the number of cores and the performance scalability of the architecture [1, 2, 3].

Current CMP designs show different organizations in their memory hierarchy. The POWER6™ [4], for instance, partially shares the second level cache among cores, while both the AMD Phenom™ [5] and the Intel® processors based on the Nehalem microarchitecture [6] share the third level cache. Besides this variety, some memory models have asymmetrical or heterogeneous characteristics. For example, the Cell processor includes a memory organization that combines the use of local memories (LMs[1]) and caches [7]. The examples above demonstrate that the optimal memory model for HPC is not defined since it highly depends on several design goals such as performance, energy consumption, scalability, area and programmability. However, all these memory models share a well-known scheme: a low-latency/high-bandwidth size-constrained stack

[1]From now on we will use LM to refer to any kind of on-chip software managed local memory.

of intermediate memories, plus a final huge high-latency/low-bandwidth main memory. In the end, one of the issues that particularly affects the achievement of the design goals is to decide how the intermediate memories are managed: implicitly by hardware, or explicitly by software.

For decades, the standard method to reduce the memory wall problem has been to bridge the gap with *hardware managed caches*. This solution is effective in improving the average performance for most applications. However, this approach may be far from the optimal solution for multi-core HPC architectures when considering performance, scalability and energy consumption all together.

The alternative model, software managed LMs, exposes the intermediate memories to the software and relies on it to orchestrate the memory operations. Usually, additional hardware support in order to perform asynchronous memory operations is required to reduce the overhead of the software management. By controlling the memory operations, the software can improve its scalability. It can also benefit from the full memory bandwidth while, at the same time, hiding the main memory latency [8, 9, 10]. Moreover, LMs consume much less energy due to the avoidance of tags and specific hardware required to maintain coherency, as well as providing much more performance due to their guaranteed access time [11]. However, to efficiently manage LMs becomes very complex when wanting to achieve all the above mentioned benefits. Therefore, this model is suitable for predictable and regular memory operations, which is usually the case of HPC applications. A clear example benefiting from the usage of LMs is the Cell. Several previous works have ported HPC applications to this platform and have achieved high performance results. In addition, new programming models and compilers (sometimes inspired by the embedded world) are contributing to hide the programming complexity [12, 13]. As a result, the software managed memory model has been already considered suitable for energy-efficient HPC, and this is shown by the fact that currently Cell-based supercomputers are at the top of the Top500 [14] and Green500 [15] rankings.

In this paper, we extend the architecture of commodity processors by adding both an LM and a very simple *programmable DMA controller* (PDC)[2] that performs asynchronous memory transfers. We demonstrate that a complex PDC is not required when the software efficiently manages the LM. Similar work has already introduced LMs (or advanced and complex PDCs) and evaluated their proposals in terms of performance [8, 9, 16, 17, 10]. In contrast, our work analyzes performance and energy consumption trends of the design space by evaluating several possible configurations. This paper has two main contributions:

- First, a hybrid memory model that combines the existing memory hierarchy with an LM plus a very simple PDC is proposed. The benefits and drawbacks of the different design options in their implementation are discussed in detail.
- Second, an LM design space exploration study for the Intel® Pentium® 4 platform is presented by analyzing performance, energy consumption and energy-delay factors for different LM sizes and memory bandwidths. Area impact and access time issues are also discussed.

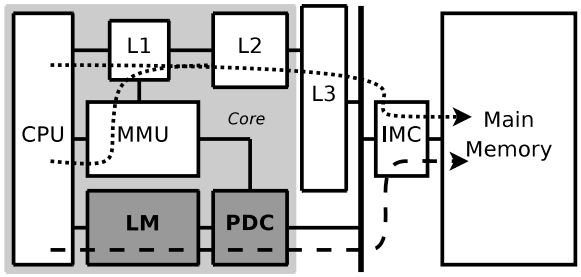

FIGURE 1. Overview of the proposed hybrid memory model. The hardware extensions, a *Local Memory* (LM) and the *Programmable DMA Controller* (PDC), are filled in dark gray. The dotted line represents the normal path to main memory, and the dashed line represents the alternative path proposed.

We have examined 27 NAS [18] computational loops in our study, and our conclusion is that for HPC applications, just a 32KB LM and a very simple PDC are required to get an average of 1.2x performance speedup factor and an energy saving of 6.21% on average when using current commodity memory bandwidths (6.4GB/s). We also show the potential of our proposal for more aggressive configurations (i.e., 256KB LM + 12.8GB/s) which show at least 2.14x performance speedup factors and energy reductions of 42.07% on average.

The rest of this paper is organized as follows: Section 2 presents the extensions we propose and discusses in detail the design alternatives and their rationale; Section 3 briefly points out the details of our compiler and run-time support. In Section 4, an overview of the methodology is presented in conjunction with two case studies and the overall results; Section 5 reviews previous work on this topic. Finally, Section 6 concludes this paper with a summary of the main research ideas and a brief outline of the future work.

2. ARCHITECTURE EXTENSIONS

This section describes the proposed architectural extensions and discusses the implications and changes required to current processor designs. Figure 1

[2] Also referred in the literature as a DMA engine, memory flow controller (MFC) or programmable memory controller (PMC).

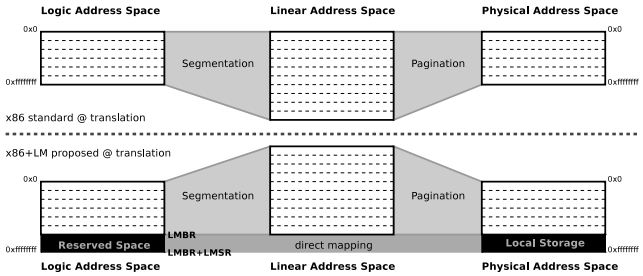

FIGURE 2. Generic and proposed AS and translation mechanisms for the x86 architecture. We propose to bypass the translation mechanisms and to direct-map logical addresses to the LM.

overviews our proposed architecture: an LM is integrated into the processor core at the same level as the L1 cache in conjunction with a programmable DMA controller (PDC) to support asynchronous data transfers from/to main memory. The PDC has access to the main memory through a bus shared with the last level of the cache hierarchy (L3 in the figure). This design enables an alternative path (the dashed line in the figure) to main memory. The dotted line in the same figure corresponds to the traditional memory access path.

One may argue that placing the LM so close to the processor could be complex since L1 is tightly integrated with the execution pipeline [10], or that it may not be worthwhile because of area limitations. However, we propose feasible and very simple modifications to the architecture. We discuss them in the following subsections.

2.1. Address space

Figure 2 shows the address space (AS) organization. A range of the logical AS is reserved and direct-mapped to the LM. The rest of the logical AS is translated to the linear AS and then virtualized using the pagination mechanism. Reserving a few kilobytes of a much bigger logical AS[3] does not affect the system usability.

Bypassing the MMU mechanisms implies that the privilege levels and the access rights are not checked when the LM is accessed. We forbid code execution (fetch instructions) from the LM and read/write access is always granted. Since the purpose of the LM is similar to a large addressable register file, we use the same privileges by analogy. This decision requires the addition of a simple check in the instruction fetcher to verify the address when an instruction is fetched.

The chosen AS layout does not require any other change to the architecture. Structures such as the load/store queue do not need to be modified as long as they work using physical addresses, which do not overlap in our configuration. In order to avoid hazards due to the usage of logical addresses, the software must assure that the linear AS does not overlap with the reserved logical AS for the LM.

Finally, following the analogy of considering the LM as a big register file, the OS should save/restore the contents of the LM when a context switch is performed. We do not evaluate the overhead of this extra save/restore operation since it is marginal in HPC environments in which CPUs are not shared, the users reserve CPUs for long periods and recently, tick-less OSes are used. This assumption may be invalid for less extreme environments; however, there are already proposals that mitigate this issue [19].

2.2. Selecting the path to memory

We need a way to differentiate memory operations between the two possible paths: cache and LM. We consider two options:

- *Extending the ISA*: Adding extensions to the current ISAs removes all the overheads associated with the path selection since the decision is taken at compile-time. For RISC architectures, this option requires the addition of a small set of memory instructions for the LM. However, for CISC architectures, in which memory operations are implicit in the addressing modes, this ISA extension implies adding a new addressing mode. If the ISA allows instruction prefixes (such as x86 ISA [20]), this issue can be reduced to the addition of just one prefix that marks the instruction as operating on the LM.
- *Base + Size registers*: Adding a couple of registers to *direct-map* a region of the logical AS to the physical LM AS. We name *LM Base Register* (LMBR) as the register to specify the LM logical start address, and the *LM Size Register* (LMSR) as the register to define the LM size. Then, in order to detect the path of the memory operation, a range check is performed for each logical address[4] generated (before starting any MMU action). The checking logic must satisfy the cycle time requirements which are highly implementation-dependent. Adding alignment constraints (e.g., LMBR aligned to LMSR boundaries) can simplify the amount of logic required and alleviates the time requirements. For example, on x86-based architectures this operation can be performed in parallel with the segmentation mechanism [20] (prior to any TLB lookup). Therefore, the cycle time is not affected. This register-based solution not only avoids ISA modifications, but it also adds flexibility to the run-time LM management (by means of modifying the registers provided).

[3] On 32bits/x86 the size of the logical AS is 4GB. Therefore, reserving, for instance, 64KB for the LM would represent about 0.001% of the logical AS.

[4] Also known as virtual address.

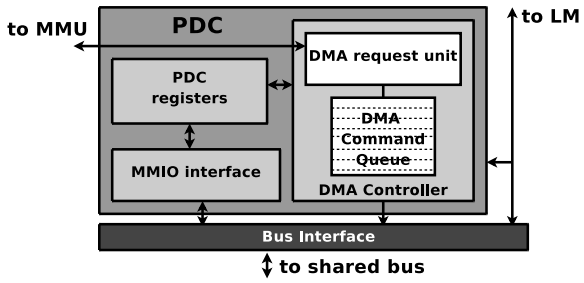

FIGURE 3. Proposed PDC block diagram.

In the end, the optimal choice is highly implementation-dependent. From now on we assume that the second mechanism is feasible on x86-based architectures, and any example given will refer to this architecture.

Finally, we only need to fork the path once we know which memory a given instruction refers to. Requests to main memory follow the standard path, address translation and L1 cache; whereas requests to LM only need to access the LM with a guaranteed latency. Besides the dynamic energy reduction of using an LM, the design proposed reduces dynamic energy consumption by avoiding frequent accesses to the translation mechanisms such as the DTLB.

2.3. Size considerations

Area constraints used to be a dominant factor in the design of new architectures. However, technology improvements allow the addition of more facilities to a processor core. Moreover, several cores can be replicated in the same chip, and several cache hierarchy levels and memory controllers are included on a single die. For instance, the Intel® processor based on the Nehalem microarchitecture can include up to 8 full-featured cores, a three level cache hierarchy, an integrated memory controller (IMC), and a Quick-Path interconnection interface for inter-processor communications on a single die [6]. As a consequence, the area required by an LM of similar capacity as the L1 cache[5] can be affordable. In fact, it has been shown that the area required for an LM is about a 34% less than that of a cache of the same capacity [11].

2.4. The Programmable DMA Controller

In our architectural proposal, we add a *programmable DMA Controller* (PDC) to manage memory operations between the main memory and the LM. The PDC has the responsibility of moving the requested data among those domains optimizing memory bus usage without affecting the execution. Figure 3 shows the PDC block diagram and its interfaces. Our PDC design is inspired by the design of the MFC present in the Cell [7], but significant simplifications have been adopted.

The main characteristic of our PDC design is simplicity. In contrast to the design of the MFC present in the Cell, which provides two interfaces to manage it (MMIO and channels), our design only keeps the memory-mapped I/O (MMIO) interface. The command values corresponding to a PDC operation (e.g., original/destination address, type command, tags...) are transmitted to the memory mapped PDC registers using non-cacheable store instructions. Once all the required arguments have arrived at the PDC, the operation is enqueued in the DMA command queue. When a DMA operation is completed, the command queue is signaled and the next operation starts. As a result, DMA operations are always executed in-order. Moreover, our PDC design only supports basic asynchronous *get* and *put*[6] operations to the LM.

In contrast to other proposals that introduce complex memory controllers [10], we have taken the decision of designing a very simple one to minimize energy consumption and complexity. We neither support gather/scatter operations [10] nor DMA lists [7], and only the memory mapped register interface is offered. Our PDC supports up to 16 pending requests as well as fixed-size data transfers. We will show that it is possible to achieve performance benefits with such simple hardware.

Before get/put operations start their execution and send requests to the bus, the PDC needs to translate logical addresses to physical addresses. The address argument referring to main memory is translated using the MMU. In contrast, the address argument referring to the LM is direct mapped using the proposed mechanism. There are no security risks as long as the access to a core's LM is only possible from the same core to which the LM is coupled. Once the addresses are translated, the PDC sends requests to the bus. In the case of a *get* operation, we can snoop the cache hierarchy to check if a recent copy of the requested data is present or we can force a flush, to main memory, of the data present in the cache hierarchy before performing the operation. The second option is used for the evaluation in Section 4. For *put* operations, the caches must invalidate the conflicting lines; this guarantees memory coherency for memory transfers. However, software must assure memory consistency by checking the completion of the PDC transfers prior to its data usage. When a command is programmed, a tag can be specified in order to check its status afterward. This mechanism resembles the one present in the Cell processor. This synchronization can be performed by polling the PDC status registers, or by using any block/wake-up mechanism. In our

[5]Power6, Intel Nehalem and AMD Phenom include 64KB, 32KB and 64KB L1 caches respectively.

[6]Get/put operation names are from the LM point of view. Therefore, a *get* operation is a transfer from main memory to the LM and a *put* operation works in the other way around.

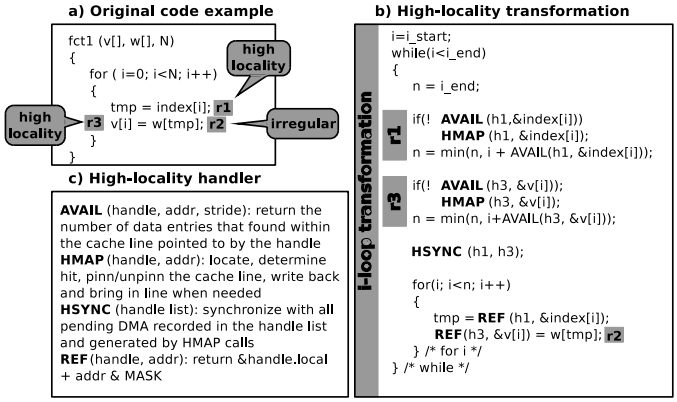

FIGURE 4. Code transformation example.

experimental framework we implement and evaluate the polling mechanism.

2.5. Main memory interface

The last level cache shares the access to main memory with the PDC. This decision intends to keep a simple design while fitting naturally in most current CMP architectures. An immediate consequence is that the same bus and protocols can be used between main memory and the caches/LM. Although this fact implies a clear advantage, it can be a bottleneck when both elements, the last level cache and the PDC, access the main memory simultaneously. However, we will show in Section 3 that the contention can be totally avoided by conscious software management.

3. COMPILER AND RUN-TIME SUPPORT

We have designed a simple run-time library similar to the one proposed by M. Gonzàlez et al. [13]. In that work, the authors present an efficient design to avoid the overheads related to the LM management. Taking that work as a reference, we have implemented the structures needed to map the regular memory accesses to the LM (what they call the *High Locality Cache*). An overview of the code transformations is shown in Figure 4. The LM management code is reduced by predicting the number of loop iterations that have their data in the LM. In more detail, the loops are modified to check the number of iterations that can be done with the data in the LM (AVAIL). New data is mapped to the LM (HMAP) if no iterations can be performed. Then, a barrier (HSYNC) is introduced prior to the computational loop which is changed to iterate over the LM data (REF)[7] . The main characteristic that should be noticed is that the contention on the shared bus is entirely avoided in this design. The reason is that the barrier (HSYNC) forces DMA transfers (HMAPs) to occur only when the execution units are not producing any requests to the cache hierarchy; no DMA transfers will take place while the inner loop is executed (and perhaps requiring access to the cache hierarchy). Finally, we rely on the compiler technology to perform such transformations automatically [21, 13].

4. EXPERIMENTAL EVALUATION

4.1. Methodology

We have evaluated a total of 27 computational loops from four different NAS benchmarks [18, 22]. The first six columns in Table 3 summarize the loops evaluated. This benchmark suite is commonly employed to evaluate HPC proposals, and by evaluating 27 different loops we can give hints about how our approach would perform in general for HPC applications. The main reason for presenting per loop results is that they allow the behaviour of the proposal to be studied for different memory access patterns (regular and regular/irregular). The overall application results follow the same trends as the ones presented because the applications evaluated are dominated by these computational loops studied.

4.1.1. Timing framework

Figure 5 overviews the experimental framework flowchart. The performance results (timing flowchart in the figure) are generated using a software cache emulator.

The loops are rewritten by performing the transformations described in Section 3 manually (inserting calls to the run-time library). In our experimental framework, the run-time library performs as it would perform in a real system except that it emulates the PDC related operations. Our approach is to map the LM on the cache hierarchy in a similar way to that used by J. Gummaraju et al. in [12, 10] (basically we ensure that the LM accesses hit the cache hierarchy). In order to validate that the LM emulator performs correctly we used a pintool [23] that simulates the memory hierarchy. Moreover, we also took memory hierarchy miss ratios using PAPI. Then, we checked that in both result sets (from PIN and from PAPI) the accesses to the LM hit the cache hierarchy (see validation flowchart in figure 5). Note that the DMA synchronization operations (the checking of the PDC status registers) do not need to be emulated for the correct execution of the benchmark.

Once we validated that the program performs as it would on a real LM-enabled platform, we need to get accurate time measurements. We read the timestamp of the processor at the start and end of the benchmark in order to get total execution time (t_{exec}). We also account for the time required to map the regions to the LM (t_{map}). Finally, we simulate the time spent in waiting for DMA operations to finish (t_{dma_wait}). For that purpose, we inserted calls to a PDC simulator when a DMA is programmed and when we perform polling on the PDC status register. In the first case, the simulator

[7]Please refer to M. Gonzàlez et al. [13] work for a complete run-time description.

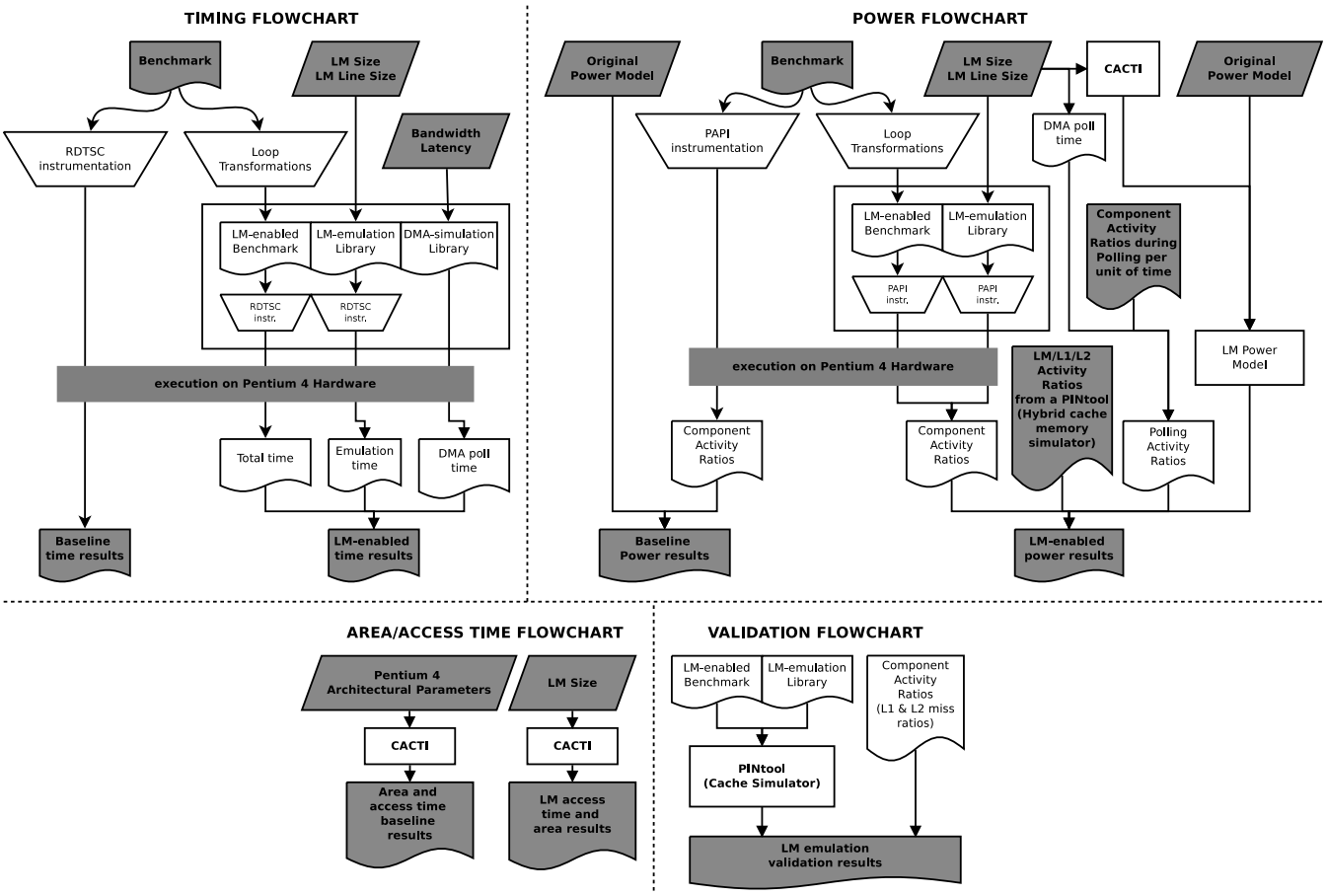

FIGURE 5. Experimental framework flowchart.

keeps track of the timestamp when the DMA operation will finish. In the second case, the simulator accounts for the time it would require to wait the DMA operation to finish (taking into account the current timestamp). The simulator derives the time required to transfer a LM line from the bandwidth, latency and the LM line size information. At the end, the simulated execution time is calculated using the following equation:

$$t_{sim} = t_{exec} + t_{dma\_wait} - t_{map} \quad (1)$$

In order to minimize the instrumentation overhead and get accurate time measurements, all the time measurements are generated using the well-known "`cpuid; rdtsc;`" instruction sequence. We have also evaluated the emulation overhead to ensure correctness. Table 1 shows that the overheads introduced by our emulation routines are minimal (compared to the time required for a single iteration of the control loop which includes all the LM management and the internal loop iterations). Moreover, they are the same order of magnitude to that the time it would require to perform the same operation in a real platform (write to the non-cacheable memory mapped PDC registers).

Finally, all applications are compiled using ICC v10.1 [24] with the following compiler flags: -03 -static and auto-vectorization disabled. The experiments are run on a 3GHz Intel® Pentium® 4 Xeon (16KB L1 8-way, 1MB L2) with 1GB RAM under FC8 (kernel version 2.6.28 with PertCtr patch [25]) and 6.4GB/s memory bandwidth in standalone mode to avoid interferences in time results.

TABLE 1. Emulation overhead.

Emulated Operation	Overhead
Program a DMA transfer	188ns/566cycles
Check a DMA to finish	266ns/678cycles

4.1.2. Power, Area and Access time framework
The power metrics (power flowchart in Figure 5) are obtained from the PMU-based power model detailed in C. Isci et al. [26]. The power model presented in that work accounts for the power consumed by the Intel® Pentium® 4 processor components, including the cache hierarchy. The main memory is not included in that model, neither in our evaluation. However, we can expect a reduction of power in the main memory due to the efficient management (reducing main memory

TABLE 2. Configurations evaluated.

	LM Size	DMA Transfer Size
Configuration-1	8KB	512B
Configuration-2	16KB	1KB
Configuration-3	32KB	2KB
Configuration-4	64KB	4KB
Configuration-5	128KB	8KB
Configuration-6	256KB	16KB
Bandwidths (GB/s)	0.8,1.6,3.2,4.8,6.4,9.6 12.8,19.2,25.6,32,64,96	

traffic). We have instrumented the applications to perform calls to PAPI [27] in order to get performance counters. The counters related to the LM, the L1 and the L2 are generated from another PINtool that simulates the hybrid memory hierarchy. The LM is added to the power model, and its power consumption is modeled using CACTI 6.0 [28]. The static LM power consumption has been added to the IDLE component, whereas the LM component (*LocalStorage* in the figures) only accounts for the dynamic power.

The area metrics and cache access time metrics (bottom left flowchart in Figure 5) are directly generated using CACTI 6.0 plus Intel® Pentium® 4 floor-plan information available in [29]. The rest of the metrics –energy and energy-delay product (EDP)– are derived from the previous ones.

4.1.3. Configurations evaluated

The baseline configuration for all the results presented in this section refers to the aforementioned platform. We do not use a more aggressive baseline (i.e., increasing the L1 as much as the extra area needed by the LM) because by enlarging the L1, we increase its access time (affecting all the programs). Moreover, we see our proposal as an extra processor feature (or functional unit) that must not affect programs which do not use it. This follows the trend that in each technology generation processors include extra features such as SIMD extensions and more functional units, but the L1 is not increased. The reason is that the L1 access time is critical to reduce the memory wall problem and to exploit data locality [30]. Thus, we study the potential gains and bandwidth requirements of adding a tightly coupled LM to the existing Intel® Pentium® 4.

We have evaluated six different LM configurations varying its size and its corresponding DMA transfer size. Table 2 summarizes the configurations evaluated. For each LM size, the DMA transfer is 1/16 of the total LM size. This ratio was chosen because it is the highest ratio in which all the benchmarks studied fit into the LM (we have evaluated other ratios, although the results are not included). Thus, for instance, the 64KB LM configuration has a PDC that transfers 4KB per command. We have simulated different memory bandwidths for each configuration in order to know the memory subsystem requirements of our solution. The main memory latency in the simulator was fixed to 135ns, which is the latency we measured in our evaluation platform.

Finally, all averages presented in this section are computed using the geometric mean which indicates the central tendency or typical value of a set of numbers reducing the outlier interferences.

4.2. Case study: 32KB LM and 6.4GB/s BW

This section details the results for a 32KB LM and 6.4GB/s memory bandwidth configuration. We have chosen this configuration because it is the most conservative one that shows speedups and energy reductions on average. We will see in Section 4.4.4 that this configuration has a faster access time than the baseline L1 and that it slightly increases the die area.

Columns 7 and 8 in Table 3 summarize the validation results of our experimental framework. The results show that most of the loops have a very high L1 hit ratio and almost 100% L2 hit ratio. The loop CG-0 is a special case due to its low L1 hit ratio. The reason is that this loop initializes five arrays at the same time. Consequently, all the data does not fit in the L1 cache, but this is not an issue since the L2 hit ratio is almost 100%. We can also observe low L1 hit ratios in IS-2, CG-6 and CG-9 loops; the fact that they contain indirect memory accesses gives us an explanation for this. Specifically, the percentage of indirect memory accesses is 50% for the loop IS-2 and 33% for the loops CG-6 and CG-9. The loop FT-3 also contains indirect memory accesses, but it still has a high L2 hit ratio because the indirect memory accesses only account for 14% of total memory accesses. The rest of the configurations studied (not detailed in the table) show slight variations in the L1 hit ratio and very similar L2 hit ratios. In conclusion, the results prove that all LM accesses hit the cache hierarchy of the experimental platform (validating the LM emulation methodology).

The last three columns in Table 3 state the speedup factor, the percentage of energy savings (the higher the better) and the percentage of reduction of the EDP (the higher the better), respectively. In general, speedups are found in all loops. Actually, the average row (the last one in the table) shows a 1.2x speedup factor. The loops IS-0, IS-1 and CG-3 get slowdowns because they are single-array initialization loops (they only write to memory). For these loops, both the extra control code executed and the DMA transfer time of our solution increase the execution time. In any case, these loops achieve speedups when using more aggressive configurations; however, these speedups are noticeably lower than the ones found in the rest of the loops. Moreover, the loops FT-0, FT-1 and FT-2 also

TABLE 3. Summary of the loops evaluated from the NAS benchmarks. Cache hierarchy validation results, speedup factors, energy and EDP percentage reductions for an LM 32KB and 6.4GB/s BW configuration.

Benchmark	Input Set	File	Function	Line	Loop #	LM 32KB and 6.4GB/s BW				
						L1 Hit	L2 Hit	Speedup Factor	Energy Savings	EDP Reduction
IS	B	is.c	rank	390	0	100%	100%	0.74x	-39.0%	-86.9%
				395	1	100%	100%	0.74x	-38.8%	-86.8%
				400	2	81.96%	74.26%	1.07x	11.24%	16.55%
				412	3	99.99%	100%	1.57x	18.68%	44.11%
MG	A	mg.c	resid	527	0	94.02%	99.97%	1.91x	37.59%	67.31%
			psinv	463	1	93.88%	99.97%	1.67x	30.28%	58.26%
			interp	684	2	97.14%	99.98%	1.66x	28.20%	56.96%
			rprj3	608	3	88.84%	99.94%	1.38x	17.17%	40.01%
			norm2u3	842	4	100%	100%	1.93x	49.24%	73.71%
FT	A	ft.c	cffts1	530	0	94.95%	99.86%	0.40x	-94.7%	-389%
			cffts2	576	1	98.37%	99.99%	0.30x	-148%	-724%
			cffts3	621	2	98.50%	99.98%	0.26x	-181%	-988%
			evolve	248	3	98.89%	99.99%	1.54x	20.22%	48.06%
CG	B	cg.c	conjgrad	386	0	39.86%	99.99%	1.66x	21.57%	52.79%
				398	1	100%	100%	1.68x	25.37%	55.63%
				480	2	99.99%	100%	1.32x	0.77%	24.81%
				488	3	100%	100%	0.47x	-86.1%	-297%
				497	4	99.99%	100%	1.72x	26.61%	57.31%
				526	5	100%	100%	1.56x	21.63%	49.81%
				554	6	83.82%	94.27%	1.08x	0.51%	8.2%
				564	7	99.99%	100%	1.26x	-3.26%	18.36%
				572	8	99.99%	100%	3.34x	57.69%	87.34%
				430	9	94.96%	91.40%	1.09x	0.67%	9.12%
				540	10	99.99%	100%	1.23x	5.25%	22.73%
				516	11	98.52%	99.99%	1.60x	24.51%	52.80%
CG	B	cg.c	main	275	12	99.99%	100%	3.68x	56.19%	88.11%
				300	13	99.99%	100%	1.21x	-6.01%	12.56%
							Average:	1.2x	6.21%	21.59%

show slowdowns. Section 4.3 explains in detail the main reason for such slowdowns.

The last two metrics, energy and EDP, show similar behaviour to the speedup metric since they are directly related to the execution time. In general, the loops with speedups show reductions in energy consumption and EDP, and vice versa. However, the loops CG-7 and CG-13 show speedups but slight increments in energy consumption. The reason is that although the execution time is reduced, the average power consumption is increased. The average power increases because there are less CPU stalls waiting for memory (more resources busy), and also because of the extra control code executed. In fact, we execute 66% more instructions on average for this 32KB LM configuration. In any case, the EDP of these two loops shows reductions. In summary, this 32KB LM + 6.4GB/s configuration presents an energy reduction of 6.21% and an EDP reduction of 21.59% on average.

4.3. Case study: Energy breakdowns

This section presents the energy breakdowns of four loops and explains them in detail. We have selected the loops that take the worst and the best profit of our proposal, as well as a case presenting both regular and irregular memory access patterns, and a representative loop of the average case. We have analyzed the loop behavior for different LM sizes with a given bandwidth of 6.4GB/s, the baseline one, and for different memory bandwidths with a given LM size of 32KB.

The results in the following figures are presented in stacked bars of energy components. The energy components are sorted to improve the readability. The static consumption (IDLE component), which basically depends on the total execution time, sits at the bottom. Then, the memory related components (LocalStorage, L1Cache, L2Cache and BUS Control) are grouped. We will see that our solution significantly reduces these components. Afterward, the components related to the branch execution (L1BPU and L2BPU)[8]are grouped. We will see that these components considerably increase their energy consumption when the polling time (the

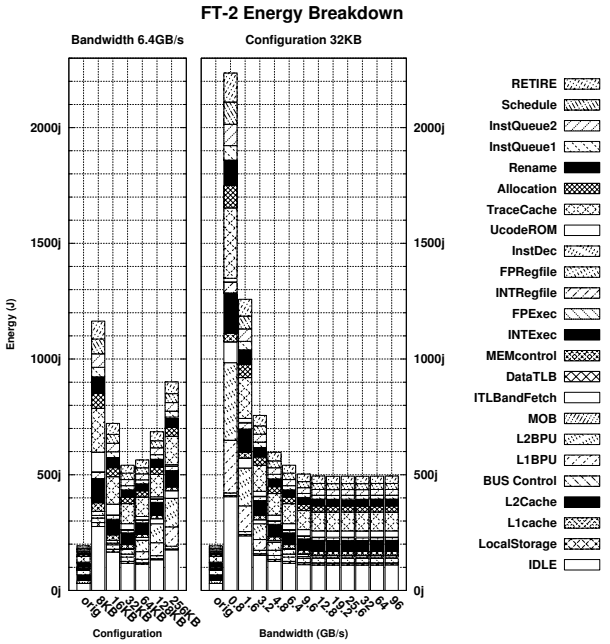

FIGURE 6. Energy breakdown of FT-2 loop.

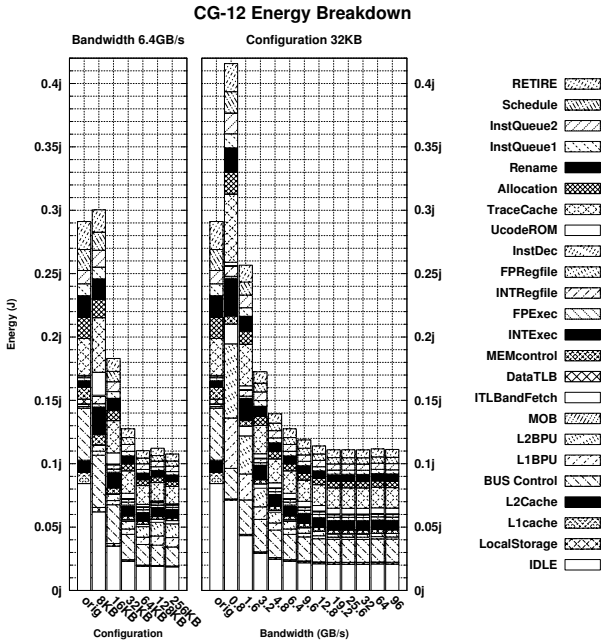

FIGURE 7. Energy breakdown of CG-12 loop.

time waiting for the completion of DMA transfers) increases. Finally, the rest of the components are depicted.

Figure 6 presents the energy breakdown of the FT-2 loop. Most of the FT loops do not benefit from our proposal. This happens because these loops are already optimized to fully exploit L1 cache locality. The FT-2 inner-most loop iterates over a data-set which fits into the L1. Furthermore, it contains a subroutine call which includes another modified loop. As a result, the LM control code overhead is substantial. This is verified in Figure 6, which shows that all components increase the energy consumption even for huge LM configurations (in which less control code is required), or for high memory bandwidths (in which no polling is needed). As a consequence, the effect of the reduction of L1 and L2 components is unnoticeable. In these cases, the compiler could anticipate this behavior and disable the code transformations. Another point to remark in Figure 6 is that energy consumption grows for LMs bigger than 32KB. The reason is that, at this point, we start to transfer useless data and, as a result, the execution time increases due to the bigger transfer sizes. This demonstrates that having LMs bigger than the working set size (not using all the LM capacity for useful data) incurs energy penalties. As stated, those loops already optimized to have a working set that fits into the cache hierarchy do not benefit of our proposal.

The case of CG-12, in Figure 7, shows the opposite behaviour. Only for very low-bandwidths or low size configurations (0.8GB/s and 8KB in the figure) the loop does consume more energy than the baseline. The reason is that the energy consumed during the polling time –waiting for DMA transfers– dominates the energy savings obtained by the reduction of L1 and L2 components. This effect is indicated by the energy increment of both BPUs components, which are directly related to the polling mechanism (polling stresses the BPUs). For the rest of the configurations and bandwidths, all components experience an energy reduction excepting the ones with big LM sizes, in which the polling causes an increase of the BPUs energy consumption once again. However, the overall energy savings predominate due to the noticeable reduction in execution time.

The previous loops have regular accesses to memory, whereas the one in Figure 8 (IS-2) contains both regular and irregular memory accesses. The trends of this loop are similar to the ones in the previous example. We can see the polling effect on the BPUs for low bandwidths again. However, in this case, the memory components are not reduced to nearly zero due to the memory operations with irregular access patterns that are not mapped to the LM. Moreover, we see that for every configuration the rest of the components consume similar energy to the baseline (resulting in less energy savings). The reason is that the execution time is not reduced substantially because this loop is dominated by the irregular accesses to memory. This is corroborated by the fact that the loop does not get benefits from higher memory bandwidths or bigger LM sizes.

Finally, Figure 9 shows the energy breakdown of the MG-3 loop, which we selected as representative of the average case trends. If we analyze the

[8]From now on, we will use Branch Prediction Units (BPUs) to refer to both L1BPU and L2BPU components.

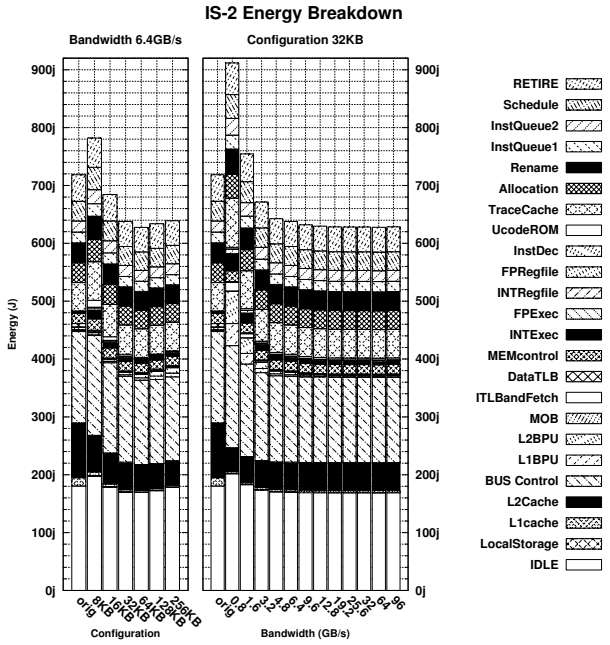

FIGURE 8. Energy breakdown of IS-2 loop.

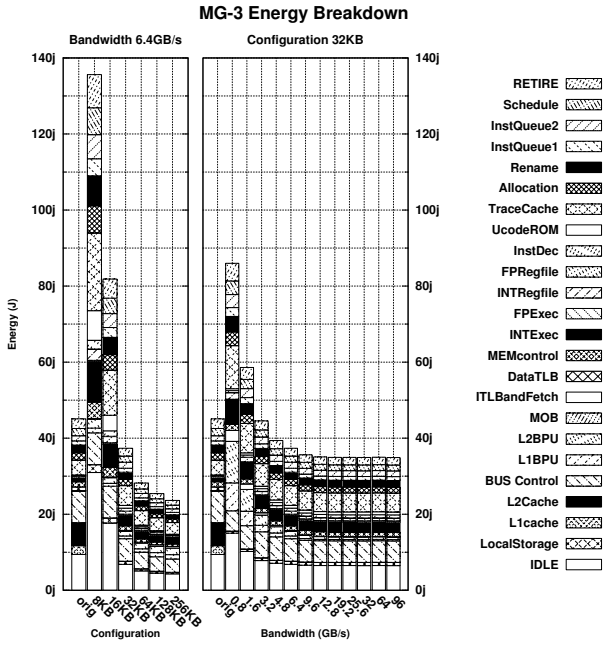

FIGURE 9. Energy breakdown of MG-3 loop.

memory bandwidth effects, the configurations with low bandwidths increase the energy consumption of the BPU components. The rest of the components also increase their energy consumption due to the higher execution times for such low bandwidths. Moreover, we see that once the benchmark reaches the point where no polling is performed, it does not get benefits of the extra memory bandwidth since it becomes compute bound. Regarding the LM size, we can see that small LMs introduce too much management code, which results in an increase in energy consumption for all components. Again, the increment of execution time is the main factor which increments the total energy consumption. In contrast, bigger LMs get benefits due to the reduction of the control code executed (and execution time) until the working set fits into the LM, which is usually not the case of HPC applications. Another detail to point out is that integer execution (INTExec) and trace-cache (TraceCache) components increase their energy consumption because they are used by the control code. In any case, for any given LM size or bandwidth, the L1 and L2 energy consumption is reduced to nearly zero, whereas the LM energy consumption still remains insignificant.

4.4. Overall evaluation

4.4.1. Performance

The average speedups for each bandwidth/LM configuration are shown at the top of Figure 10. We see that the smallest configurations, the 8KB and 16KB sizes, get slowdowns for any bandwidth. The main reason for this is that the control code overhead dominates the performance gains of using an LM. For larger LM configurations, we start getting performance benefits among the range [3.2-4.8]GB/s, which is the point where DMA transfers start to be efficiently overlapped with the control code. Moreover, the bigger an LM configuration is, the less control code is executed. That fact results in higher speedups for bigger LM configurations.

Another remarkable characteristic is the convergence point of each LM configuration. The figure shows that the smaller LM configurations become compute bound (not getting benefits of extra memory bandwidth) much faster than the bigger ones. The reason is that bigger configurations require much more memory bandwidth in order to overlap efficiently their bigger DMA transfers with the control code execution.

4.4.2. Energy

The average energy results are depicted in the center of Figure 10. Energy consumption behaves in a similar fashion (showing the same trends) as performance because it is directly related to the execution time. Again, the smaller configurations do not get energy savings for any memory bandwidth, and bigger LM configurations exploit the available memory bandwidth better. In this case, the four bigger configurations start getting benefits at 4.8GB/s. We do not obtain the same factors as performance since our solution executes many more instructions (that consume energy) in less time (thus requiring more power). Actually, the average power consumption increases because there is more activity in the processor (less processor stalls due to memory). However, this increment is low due the introduction of the LM. Therefore, we conclude that the improvement in energy consumption is mainly due to the improvement in performance, and that the LM

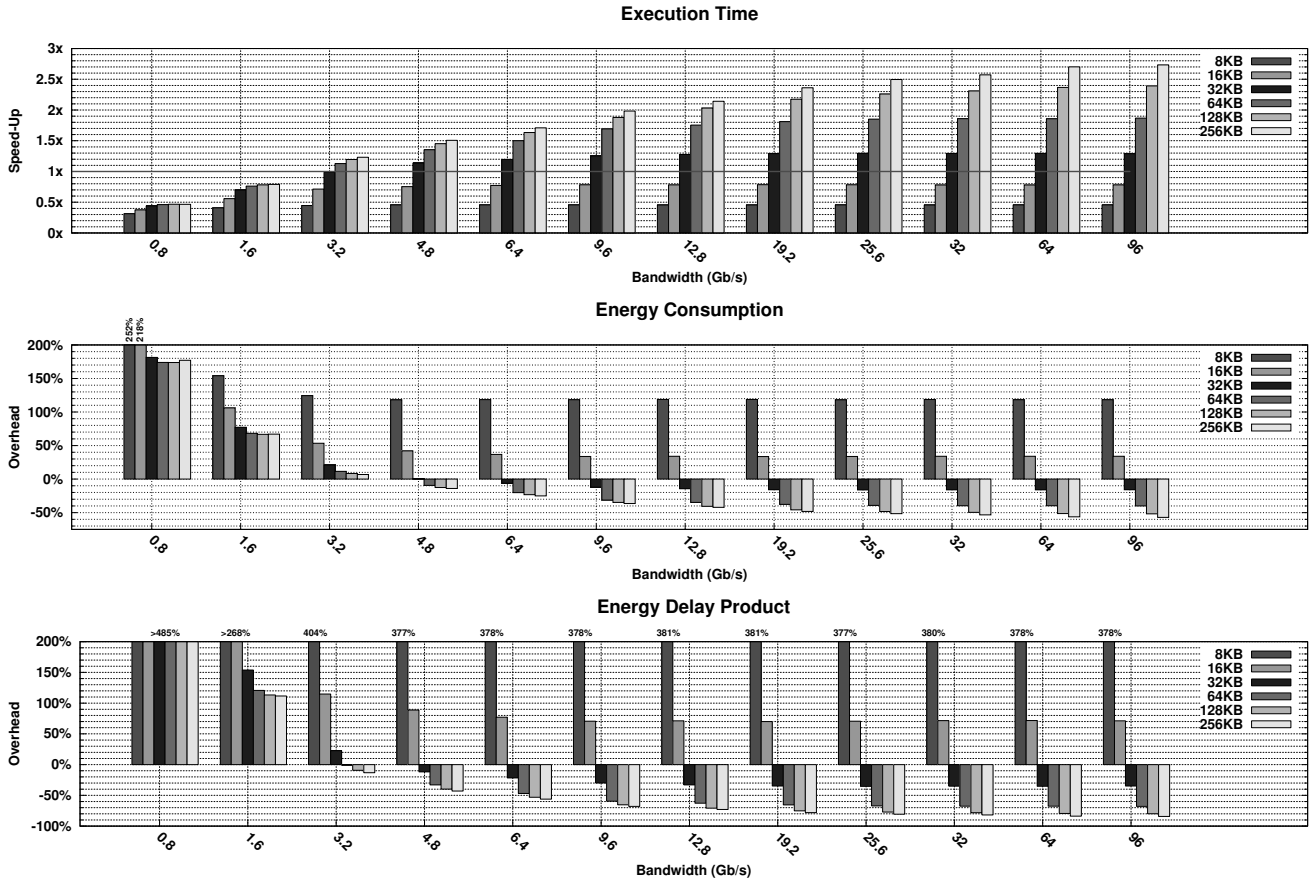

FIGURE 10. Average speedup factor, energy consumption and EDP for each configuration and bandwidth.

allows us to maintain the average power consumption although the activity in the processor is much higher.

4.4.3. Energy Delay Product

At the bottom of Figure 10, both metrics already analyzed are put together using the energy delay product (EDP) metric. Consequently, we see the very same tendencies. However, the overheads and speed-ups are emphasized since in most of the cases, when we get benefits on performance we also get benefits in energy consumption and the other way around. We only studied one case in which we do not get benefits or overheads in performance and energy at the same time. For an LM of 32KB and 4.8GB/s memory bandwidth, we get 1.14x in performance, but 0.34% increment in energy consumption. As a result, we approximately get a 11.38% reduction in the EDP. This evidence corroborates the fact that most of the energy savings come from the improvements in execution time (performance).

4.4.4. Area and Access Time Considerations

Table 4 summarizes the total area and access time increments for each configuration. The baseline configuration is the one of the Intel® Pentium® 4 used during the evaluation. We do not take into account the area required by the PDC as it is marginal compared to the one required by the LM. Obviously, the bigger the LM is, the more it increases in area size and access time. However, the access time required for small LM configurations is much shorter than that required for the baseline one (the 16KB 8-way L1 cache present in our Intel® Pentium® 4). In short, the optimal configuration regarding both parameters is a trade-off between the performance and the area impact. We consider that the LM configurations up to 64KB are feasible nowadays since their impact in area and access time are affordable. Moreover, we believe that the bigger ones can be also affordable in the near future or when considering reductions of the last level of the cache hierarchy.

4.5. Overall Results

In previous sections we evaluated performance, energy and EDP metrics. Additionally, we provided a brief discussion about the area and LM access time issues. Figure 11 relates all the metrics to each other for the LM configurations that get performance and energy benefits. The horizontal line at 100% denotes the baseline configuration (our evaluation platform). The

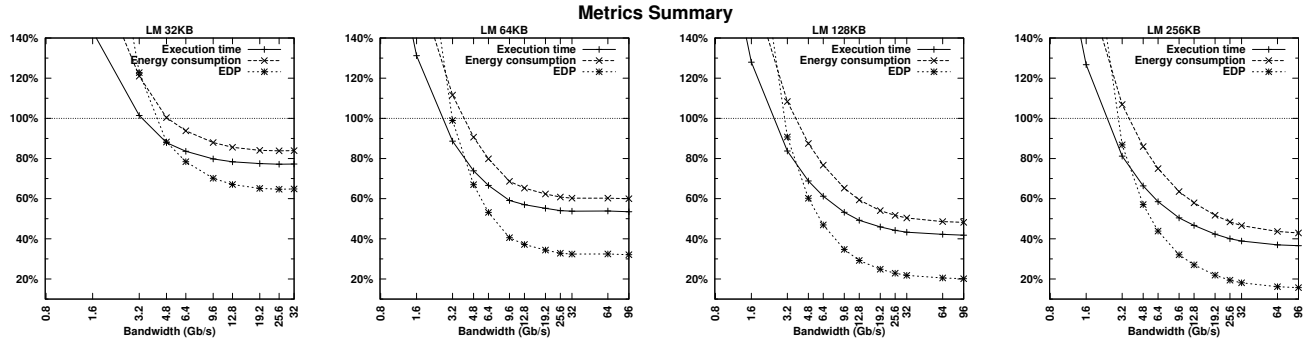

FIGURE 11. Normalized overall metrics summary. The lower the better.

TABLE 4. Area and access time increments.

LM Configuration	% Area Increment	% Access Time Increment
8KB	0.73%	-64.18%
16KB	1.21%	-59.8%
32KB	5.09%	-0.17%
64KB	8.35%	3.76%
128KB	17.79%	12.33%
256KB	31.28%	24.27%

results of the four configurations show that if we maintain the same EDP as the baseline (100% line), we get performance speedups and energy overheads. The 32KB LM configuration starts getting benefits when the bandwidth is around 4.8GB/s. The other three configurations require even less memory bandwidth (around 3.2GB/s). Nevertheless, the important point to remark from these figures is that the bandwidth requirements needed to start reducing both energy and execution time are common in current commodity processors. Actually, they are far less than that available for the baseline configuration (6.4GB/s).

In the end, we conclude that the most restrictive issues are area and LM access time, which are very implementation dependent. We demonstrated that only a 32KB LM is required, with commodity memory bandwidth configurations, to get improvements in performance and in energy consumption while still keeping a very low impact in area and access time. We believe that even bigger LM configurations can be affordable in current CMPs design restrictions, which sometimes spend more than half of the die area in the cache hierarchy.

Moreover, we show the potential of our solution for higher bandwidths, as well as less restrictive area and access time requirements. The most size-aggressive configuration, a 256KB LM, gets a 2.14x speedup factor and an energy reduction of 42.07% on average requiring only a memory bandwidth of 12.8GB/s. Higher speedups and energy savings are obtained for higher memory bandwidths.

5. RELATED WORK

The introduction of an on-chip LM in a core is not novel, but very recent. Some research studies have been made around this topic, but we have not found any that explore a hybrid approach in which the LM is private to the core, thus only addressable from it, and targeting general purpose processors.

Some of the related works take a drastic approach in which the traditional cache hierarchy is replaced by LMs, and a streaming programming model is introduced. For instance, J. Levich et al. [1] compare the two models (LM vs. caches) in terms of performance and energy consumption. In that work, the authors conclude that both models perform very similarly, but they show different efficiency levels concerning bandwidth utilization and energy consumption. In the study of the streaming model no automatic code transformations are presented, nor is any compiler/runtime support. This is an important point, since our method does not require any code rewriting effort. In our proposal, current available compiler technology is used to exploit the LMs in the HPC domain. Finally, another important aspect is not considering the possibility of having a hybrid scheme, which corresponds to our proposal.

Other works, that study the use of LMs, target specific domains. For instance, O. Unsal et al. [31] propose a cache architecture (Cool-cache) for the multimedia domain. Media applications are profiled to detect their most common access patterns, and it is from this information that the cache architecture is designed. The cited authors propose a software managed cache based on the same hypothesis: memory access patterns are predictable, and thus it is possible to orchestrate all memory operations in software with a reasonable impact on performance. We test the same hypothesis but in the HPC domain and with remarkable differences in the design: simplicity and no profiling

information. Our design only needs both the compiler support and the run-time to efficiently manage the LM.

J. Gummaraju et al. [10] introduce similar ideas but using different techniques. In that work, the authors propose to modify general purpose processors for supporting stream programming models. The second level cache is partitioned, so that a selected area is used as an LM. The hardware pre-fetcher is extended to support programmable asynchronous memory transfers (between the emulated LM and main memory) and gather/scatter operations. We believe that the cited solution is more complex than our proposal. Another specific drawback of that work is using the L2 as an LM. This fact reduces the chances for reducing the energy consumption since memory accesses still trigger the logic associated to cache accesses (e.g., tag check). This is somehow losing the opportunities that the stream programming models give. Introducing a separate LM managed by software reduces the energy consumption of many, if not all, memory accesses which follow an access pattern that justifies the introduction of a stream programming model.

Other related works have been focused on improving the memory controller for taking profit of predictable access patterns. Yamada et al. [8] propose a hardware and software technique for data relocation and pre-fetching that improves cache performance. Another example is the work of Zhang et al. [17], which introduces the Impulse Memory Controller. That advanced memory controller adds an optional level of address indirection at the memory controller which provides to the applications the control of how their data is accessed and cached. Both works improve performance but they do not get the energy benefits of using an LM. Finally, McKee et al. [9] propose the Stream Memory Controller (SMC) system which combines compile-time detection of streams and run-time selection of the access order. They use buffers to pre-fetch memory streams that can be seen as our LM. However, they model the buffers as FIFO and, as a result, their proposal only targets in-order processors. Apart from that their model is less flexible than ours, they do not study possible energy benefits.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we studied the implications of extending commodity processors with a hybrid memory model. Specifically, we proposed to extend the general purpose processors with a local memory (LM) and a very simple programmable DMA controller (PDC) that performs asynchronous memory transfers. We discussed in detail the benefits and the drawbacks of different design options. We also performed an LM design space exploration study for a Intel® Pentium® 4 platform. We presented results for performance, energy and energy-delay factors for different memory bandwidths and LM sizes. Area impact and access time issues were also discussed. A total of 27 NAS computational loops have been analyzed in our study as a representative of HPC applications and we concluded that, for such applications, only a 32KB LM plus a simple PDC are required to get a 1.2x performance speedup factor and an energy saving of 6.21% on average when using current commodity memory bandwidths (6.4GB/s). We have shown the potential of our proposal for more aggressive configurations (i.e., 256KB LM + 12.8GB/s), which presented at least 2.14x performance speedup factors and 42.07% energy savings on average. We believe that these bigger LM configurations can be affordable in current CMPs designs, which sometimes spend more than half of the die area in the cache hierarchy.

Besides, this work opens the possibility of considering a reduction of some levels of the cache hierarchy (L2/L3). Specifically, we are exploring the possibility of adding an LM and reduce the last level of the cache hierarchy in order to get benefits on performance and energy consumption while maintaining the same area requirements.

FUNDING

This work was supported by the Ministry of Science and Innovation of Spain (CICYT) [TIN-2007-60625] and the Generalitat de Catalunya [2009-SGR-980].

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their comments that helped us significantly improve the presentation of our work. We are also indebted to the colleagues of our department and research group for their helpful feedback. The main author would like to specially thank Carlos Villavieja, for his valuable help and guidance as well as to acknowledge the support of his funding body, Barcelona Supercomputing Center (BSC).

REFERENCES

[1] Leverich, J., Arakida, H., Solomatnikov, A., Firoozshahian, A., Horowitz, M., and Kozyrakis, C. (2007) Comparing memory systems for chip multiprocessors. *SIGARCH Computer Architecture News*, **35**, 358–368.

[2] Murphy, R. (2007) On the effects of memory latency and bandwidth on supercomputer application performance. *IISWC '07: Proceedings of the 2007 IEEE 10th International Symposium on Workload Characterization*, Boston, MA, USA, 27-29 September, pp. 35–43. IEEE Computer Society, Washington, DC, USA.

[3] Kandemir, M., Ozturk, O., and Karakoy, M. (2004) Dynamic on-chip memory management for chip multiprocessors. *CASES '04: Proceedings of the 2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, New York, NY,

USA, 22-25 September, pp. 14–23. ACM, Washington, DC, USA.

[4] Le, H., Starke, W., Fields, J., O'Connell, F., Nguyen, D., Ronchetti, B., Sauer, W., Schwarz, E., and Vaden, M. (2007) IBM POWER6 Microarchitecture. *IBM Journal of Research and Development*, **51**, 639–662.

[5] AMD 2006 Technology Analyst Day: Official Introduction of K10 Microarchitecture (slides). http://www.amd.com/us-en/assets/contenttype/downloadableassets/ philhesteram-danalystdayv2.pdf. Retrieved 4-9-09.

[6] 320835-002), D. N. (2008). Intel Core i7 processor extreme datasheet.

[7] Kahle, J. (2005) The Cell processor architecture. *MICRO 38: Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, Barcelona, Spain, 12-16 November, pp. 3–4. IEEE Computer Society, Washington, DC, USA.

[8] Yamada, Y., Gyllenhall, J., Haab, G., and Hwu, W. (1994) Data relocation and prefetching for programs with large data sets. *MICRO 27: Proceedings of the 27th Annual International Symposium on Microarchitecture*, San Jose, California, United States, 30 November - 2 December, pp. 118–127. ACM, New York, NY, USA.

[9] McKee, S. A., Wulf, W. A., Aylor, J. H., Salinas, M. H., Klenke, R. H., Hong, S. I., and Weikle, D.A.B. (2000) Dynamic access ordering for streamed computations. *IEEE Trans. Comput.*, **49**, 1255–1271.

[10] Gummaraju, J., Erez, M., Coburn, J., Rosenblum, M., and Dally, W. J. (2007) Architectural support for the stream execution model on general-purpose processors. *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, Pereslavl-Zalessky, Russia, 3-7 September, pp. 3–12. IEEE Computer Society, Washington, DC, USA.

[11] Banakar, R., Steinke, S., Lee, B.-S., Balakrishnan, M., and Marwedel, P. (2002) Scratchpad memory: Design alternative for cache on-chip memory in embedded systems. *CODES '02: Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, Estes Park, Colorado, 6-8 May, pp. 73–78. ACM, New York, NY, USA.

[12] Gummaraju, J. and Rosenblum, M. (2005) Stream programming on general-purpose processors. *MICRO 38: Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, Barcelona, Spain, 12-16 November, pp. 343–354. IEEE Computer Society, Washington, DC, USA.

[13] Gonzàlez, M., Vujic, N., Martorell, X., Ayguadé, E., Eichenberger, A. E., Chen, T., Sura, Z., Zhang, T., O'Brien, K., and O'Brien, K. (2008) Hybrid access-specific software cache techniques for the cell be architecture. *PACT '08: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, Toronto, Ontario, Canada, 25-29 October, pp. 292–302. ACM, New York, NY, USA.

[14] Top 500 Supercomputer Sites List. November, 2008. http://www.top500.org/lists/2008/11.

[15] The Green500 List. November, 2008. http://www.green500.org/lists/2008/11/.

[16] Okawara, H. (2000) SCIMA: Software controlled integrated memory architecture for high performance computing. *ICCD '00: Proceedings of the 2000 IEEE International Conference on Computer Design*, Austin, Texas, USA, 17-20 September, pp. 105–114. IEEE Computer Society, Washington, DC, USA.

[17] Zhang, L., Fang, Z., Parker, M., Mathew, B. K., Schaelicke, L., Carter, J. B., Hsieh, W. C., and McKee, S. A. (2001) The impulse memory controller. *IEEE Trans. Comput.*, **50**, 1117–1132.

[18] Bailey, D. H. and et al. (1991) The NAS parallel benchmarks—summary and preliminary results. *Supercomputing '91: Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Albuquerque, New Mexico, United States, 18-22 November, pp. 158–165. ACM, New York, NY, USA.

[19] Pyka, R., Fasbach, C., Verma, M., Falk, H., and Marwedel, P. (2007) Operating system integrated energy aware scratchpad allocation strategies for multiprocess applications. *SCOPES '07: Proceedings of the 10th International Workshop on Software & Compilers for Embedded Systems*, Nice, France, 20 April, pp. 41–50. ACM, New York, NY, USA.

[20] Intel 64 and IA-32 Architectures Software Developer's Manual : Instruction Set Reference. www.intel.com.

[21] Chen, T., Zhang, T., Sura, Z., and Tallada, M. G. (2008) Prefetching irregular references for software cache on Cell. *CGO '08: Proceedings of the Sixth Annual IEEE/ACM International Symposium on Code Generation and Optimization*, Boston, MA, USA, 6-9 April, pp. 155–164. ACM, New York, NY, USA.

[22] Npb2.3-omp-c.tgz . http://www.hpcs.cs.tsukuba.ac.jp/omni-openmp/download/download-benchmarks.html.

[23] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. (2005) PIN: Building customized program analysis tools with dynamic instrumentation. *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Chicago, IL, USA, 11-15 June, pp. 190–200. ACM, New York, NY, USA.

[24] Intel C++ Compiler 10.1 Professional Edition for Linux. www.intel.com.

[25] PerfCtr v2.6.28 http://user.it.uu.se/mikpe/linux/perfctr.

[26] Isci, C. and Martonosi, M. (2003) Runtime power monitoring in high-end processors: Methodology and empirical data. *MICRO 36: Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, San Diego, CA, USA, 3-5 December, pp. 93–104. IEEE Computer Society, Washington, DC, USA.

[27] Performance Application Programming Interface. http://icl.cs.utk.edu/papi/.

[28] Wilton, S. J. E. (1996) CACTI: An enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits*, **31**, 677–688.

[29] Isci, C. and Martonosi, M. (2003) Runtime power monitoring in high-end processors: Methodology and empirical data. Technical report. Princeton University Electrical Engineering Department, Princeton, NJ, USA.

[30] Hennessy, J. L. and Patterson, D. A. (2002) *Computer Architecture: A Quantitative Approach (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann, San Francisco, CA, USA.

[31] Unsal, O. S., Ashok, R., Koren, I., Krishna, C. M., and Moritz, C. A. (2001) Cool-cache for hot multimedia. *MICRO 34: Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, Austin, Texas, 1-5 December, pp. 274–283. IEEE Computer Society, Washington, DC, USA.