Center for Embedded Computer Systems
University of California, Irvine

SPMPool:
Runtime SPM Management for Embedded Many-Cores

Hossein Tajik, Bryan Donyanavard, Janmartin Jahn, Joerg Henkel, Nikil Dutt

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-2620, USA

{tajikh, bdonyana, dutt}@uci.edu, {janmartin.jahn, henkel}@kit.edu

CECS Technical Report 14-08
May 12, 2014

SPMPool:
Runtime SPM Management for Embedded Many-Cores

Hossein Tajik[1], Bryan Donyanavard[1], Janmartin Jahn[2], Joerg Henkel[2], Nikil Dutt[1]
[1]Center for Embedded Computer Systems, University of California Irvine
[2] Chair for Embedded Systems, Karlsruhe institute of technology
{tajikh, bdonyana, dutt}@uci.edu, {janmartin.jahn, henkel}@kit.edu

ABSTRACT
Distributed scratchpad memories (SPM) in embedded many-core systems require careful selection of data placement such that good performance can be achieved. In this paper, we propose SPMPool to share the available on-chip scratchpads on many-cores among executing applications in order to reduce the overall memory access latency. By pooling SPM resources, we can assign underutilized memory resources, due to idle cores or low memory-usage, to applications based on their needs. As the first workload-aware SPM mapping solution for many-cores, SPMPool dynamically allocates data at runtime in order to address the unpredictable set of applications concurrently executing (workload); this is accomplished without prior knowledge of the workload. The data mapping can be adaptively re-allocated at runtime based on temporal variance of the workload and the proximity of SPM resources. Our simulations for many-core systems with a range of configurations from 16 to 1024 cores show that significant improvements can be achieved: for many complex application scenarios, SPMPool can decrease the total memory access latency by up to 55% compared to limiting executing cores to use their local SPM.

1. INTRODUCTION
The ongoing trend towards embedded single-chip many-core systems is largely driven by the physical and economic limits of single-core systems (e.g. the power wall). Embedded many-core systems increase their performance within a reasonable power envelope by offering highly parallel computational resources. This allows embedded systems to run demanding applications such as real-time video processing, live object tracking, etc. Typically, SPMs are deployed as on-chip memories in embedded many-cores for achieving lower power and better predictability. Such a large number of SPMs distributed over many cores makes memory mapping challenging for the controlling software.

In order to store the most frequently used data in SPM while rarely used data may remain in off-chip memory, the memory accesses must be accurately predicted when allocating data. Dynamic scenarios can result in memory accesses that are very hard to predict when data is allocated. Additionally, in multi- and many-core systems, the temporal variance of a single application's memory access pattern can result in excessive accesses to off-chip memory. In order to permit such dynamic scenarios in which applications start and stop at any time, data allocation needs to be adapted at runtime.

The abundant presence of processing elements in many-cores results in periods of execution in which not all cores are utilized. In this case, the unutilized memory resources associated with idle cores present an opportunity for more efficient use of on-chip resources. Furthermore, the high variability of memory intensity between concurrently executing applications causes on-chip resources to be more valuable for some applications over others.

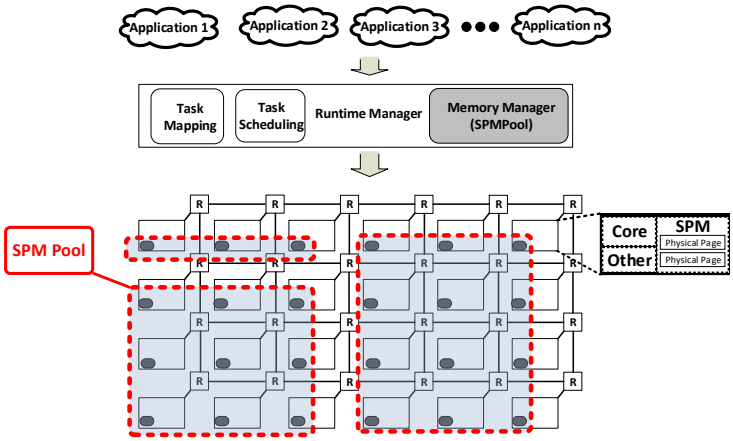

Figure 1. SPMPool role in many-core system

Existing SPM management schemes for multi- and many-cores either extend uni-core SPM mapping techniques without further consideration by limiting cores to use their local SPMs only, or propose mappings with the assumption that the workload is predictable at compile time, which is unrealistic. In order to address these issues and opportunities, this paper presents SPMPool for managing systems that integrate hundreds of cores with on-chip scratchpad memories, and off-chip memory. SPMPool, as part of a runtime system, manages memory mapping and accessing of different applications in many-core systems (Figure 1). The SPMs are shared among cores, and on-chip memory is allocated adaptively between applications based on their memory requirements. The objective of our SPMPool method is to maximize the performance (by minimizing memory access latency). By accounting for dynamic application scenarios, our approach can significantly outperform the state-of-the-art memory management schemes while offering a high degree of scalability even for large systems and many concurrently executing memory-intensive applications. As a result, SPMPool now permits dynamic and increasingly unpredictable application scenarios in systems with scratchpad memory.

To summarize, our novel contributions are:

1) An architecture that enables sharing ("pooling") of SPMs among concurrently executing applications with varying memory needs.
2) Techniques to allocate data adaptively when applications are started or stopped.
3) Illustration of the significant impact of temporal memory access variation within an application on memory placement.

In Section 2 we present a small motivational example to demonstrate potential benefits of SPM sharing. Sections 3 and 4 outline the base architecture for SPMPool and the memory mapping formal problem definition. Section 5 describes the

SPMPool architecture including all enabling software and hardware components. In Section 6 we present our experimental setup and results. Section 7 includes further results to demonstrate the importance of temporal variance of memory accesses within a single application. In Section 8 we discuss the different overheads of implementing SPMPool. Finally, Section 9 overviews related work and positions SPMPool as a new contribution.

2. MOTIVATION

In many scenarios, the amount of data used by applications exceeds the amount of SPM resources on-chip. In order to avoid severe performance degradation, the subset of data stored in SPM must be carefully selected. The state-of-the-art mechanisms for allocating data to SPM select this subset based on the size of the SPM available at the core on which the respective application is executing. However, if there are fewer applications than cores, the SPM resources belonging to unused cores are unutilized. Another form of underutilization of SPM resources occurs when concurrently executing applications have considerably different levels of memory usage.

In both cases, severe performance penalties can result from inefficient SPM usage. A solution to this problem could be the redistribution of SPM resources to executing applications with the goal of reducing the overall memory latency of all applications combined. To exemplify, let us consider a multi-core architecture with a 4-core mesh network as illustrated in Figure 2.

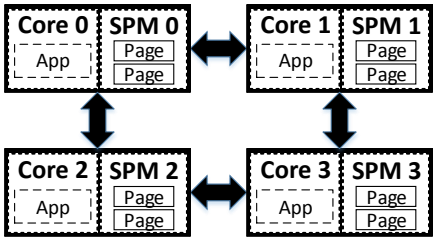

Figure 2. Example of a system with four cores and four SPMs

Each core has an SPM with capacity of two pages. Now consider the sample workload shown in Figure 3 consisting of three applications with a working set of four virtual pages each. All three applications start executing concurrently.

Application X	App A	App B	App C
Page 0# accesses	Page 0 75	Page 0 1000	Page 0 5
Page 1# accesses	Page 1 100	Page 1 900	Page 1 5
Page 2# accesses	Page 2 50	Page 2 900	Page 2 5
Page 3# accesses	Page 3 5	Page 3 1000	Page 3 10

Figure 3. Applications and their working set of pages, along with the total number of accesses to each page over the entire execution

Figure 4 shows the resulting mapping for this working set on the system using each application's local SPM only. In this case, many highly accessed pages are relegated to off-chip memory, causing accesses to them to be costly, while one SPM is unused completely.

The total memory access latency according to Table 1 for Application A is 5125 cycles, Application B is 164000 cycles, and App C is 915, resulting in a total memory access latency of 170040 cycles after all applications complete their execution. A superficial inspection of this memory mapping reveals that Application B is forced to place two highly utilized memory pages, Page 1 and Page 2, in off-chip memory, while SPM pages in Tile 3 were unused. Also, Application A has placed one highly utilized memory page, Page 2, in off-chip memory, while Application C has used one SPM slot for a relatively low utilized page. If we expanded the pool of available memory for each application by allowing them to use non-local SPMs, we could place memory pages on-chip according to their utilization and use all SPM resources. Using this approach, the alternative configuration shown in Figure 5 successfully reduces the total amount of off-chip memory accesses for all applications to 11390. In this example, and through the rest of the paper, we assume all SPMs combined create a single pool.

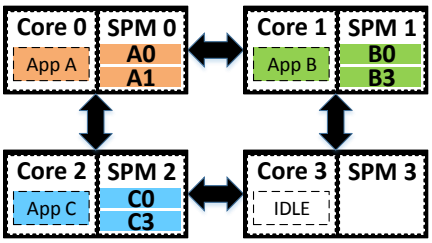

Figure 4. Traditional mapping of application memory to local SPMs

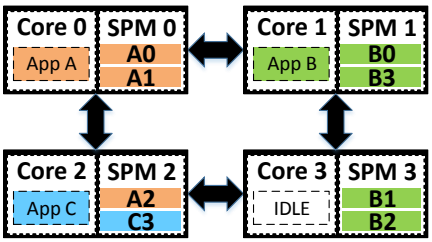

Figure 5. Alternative memory configuration for the same workload

Table 1. Cost of Access (Number of Cycles) to each memory

	SPM 0	SPM 1	SPM 2	SPM 3	Off-chip
Core 0	1	4	4	7	90
Core 1	4	1	7	4	90
Core 2	4	7	1	4	90
Core 3	7	4	4	1	90

We propose to combine all SPMs in this manner as a global SPM pool that provides memory resources accessible to any executing application. Through global sharing of physical SPMs, SPMPool aims to minimize the overall access latency incurred by memory accesses. In order to accomplish this we must provide a mechanism to determine the memory mapping based on the relative utilization of all memory pages in use.

Current allocation methods either completely ignore any potential contention for memory resources from other applications, or assume the set of executing applications at any given time is predictable. When an application is started or stopped, previously unused data may be accessed frequently and data that has previously been allocated to SPM may no longer be accessed. Hence, the state-of-the-art techniques can hardly avoid severe performance penalties in such scenarios. SPMPool sets out to address these issues that arise when applications start or stop at

any time and may execute concurrently. Sections 5 and 6 explore the opportunities presented by these shortcomings.

Additionally, in multi- and many-core systems, current SPM memory allocation techniques do not address the temporal variance of a single application's memory access pattern. For example, Figure 6 extends our original example to illustrate a sample access pattern for a single application, Application A, over its course of execution.

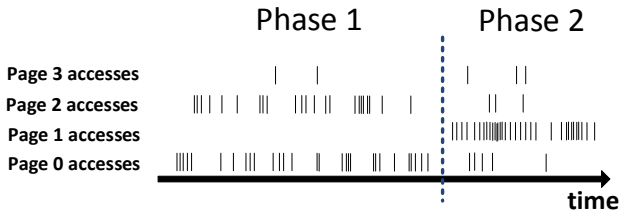

Figure 6. Temporal Variance of memory access pattern for a single application (Application A)

If we don't consider the temporal variance of accesses, Page 0 and Page 1 will be placed in the SPM (Figure 7a), as we saw in Figure 4. This will cost 5125 cycles of memory access latency for Application A. But if we split the execution time into two distinct phases (separated by the dotted line in Figure 6), we'll have two different memory mappings for each phase (Figure 7b). In the first phase, Page 0 and Page 2 will be placed in the SPM; while Page 1 and Page 3 will reside in the SPM during the second Phase. The memory access latency attributed to Application A becomes 1120 cycles with the increased cost of one memory remapping. The state-of-the-art techniques typically only load an application's pages into SPM when the application starts, which is ineffective for such variant access patterns. The effect of this temporal variance is further explored in Section 7.

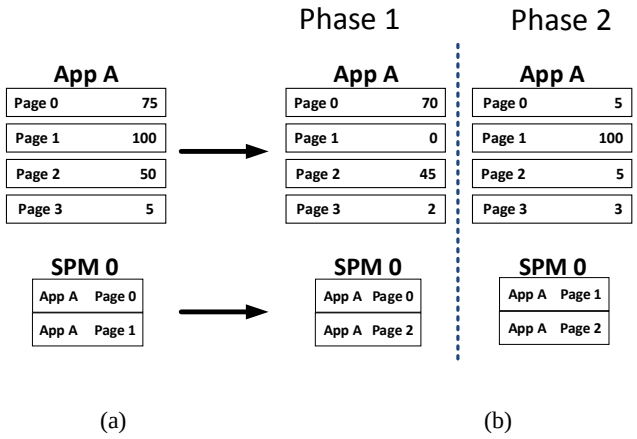

Figure 7. Page list and SPM mapping for Application A (a) before and (b) after accounting for phasic behavior

3. ARCHITECTURE DESCRIPTION

In this paper, we target tiled many-core systems in which tiles are connected via a mesh-like Network on Chip (NoC). The memory subsystem is distributed regularly among all tiles. Each tile consists of a core, SPM, MMU, and network interface to manage NoC traffic for inter-tile communication (Figure 2).

In our model, an application consists of a single thread that executes atomically on its assigned core. Each application has a private virtual address space not shared with any other application.

The major events that encapsulate the execution of each application are (1) application start and (2) application stop. Any application may experience one of these events at any time. Because each application runs to completion, (1) and (2) are experienced exactly once by all applications.

At runtime, application scheduling, mapping, and memory management is performed by a central runtime manager (Figure 1). Upon entry into the system, an application is initially mapped to a free core randomly by this central controlling instance, and the SPM memory mapping is updated with the appropriate memory pages fetched from main memory. When an application is stopped, all of its pages stored in on-chip memory are invalidated and written back to main memory and the core is freed. The next sections define in detail the challenges presented by mapping the many pages of a dynamically changing workload into the limited amount of on-chip memory resources, and specify the augmenting hardware and software to enable our solution to these challenges, SPMPool.

4. PROBLEM DEFINITION

The general goal of SPMPool is to select a subset from many application pages and assign them to the limited number of on-chip memory slots in order to get minimal access latency. To implement this scheme it is necessary to characterize the memory behavior of each application. For this work, we simplify this characterization by assigning each page a single value that represents the number of times that page is accessed. Based on these values, we determine which pages are most useful for each application. Once all applications are characterized by values for all their pages, a physical to virtual memory allocation can be generated (memory mapping). By comparing the values of each page in the set of all executing applications, a memory map can be generated that prioritizes on-chip physical memory resources for the most utilized pages.

If we assume all the architectural requirements for SPM sharing (described in Section 3) are available, the best memory mapping can be expressed as an optimization problem. The pool of SPMs itself is specified by the number of SPMs, their size, and the arrangement of tiles. Provided the pool information, we should know the existing mapping for each application, as well as the memory access latency from each tile to each SPM. The optimization problem should generate a virtual to physical memory mapping for any point in time to minimize the overall access latency caused by memory accesses of all applications. The optimization problem should be executed dynamically at runtime, with the help of information generated statically.

Let *APP* be the set of executing applications and *SPM* be the set of SPMs on the chip. Each application has a set of pages, *P*; the goal is to map each page to one of the SPMs or to the off-chip memory, such that overall performance of the system is maximized. We assume that application mapping is known and therefore each application is mapped to a core. If we consider any time, the following optimization problem should be solved:

Inputs:

APP	*set of application*
SPM	*set of SPMs*
Page	*set of pages for each application*
Cap[i]	*capacity (number of pages) in SPM[i]*
Cost[i, j]	*cost of access from APP[i] to SPM[j]*

Access[i, j] — *number of accesses to App[i].PAGE[j] over the execution of APP[i]*

Output:

$$\forall a \in APP, \forall p \in a.PAGE, \forall s \in SPM:$$

$$Map[a, p, s] = \begin{cases} 1 \\ 0 \end{cases}$$

The desired output is *Map[a, p, s]*: If *Map[a, p, s]* = 1, Page *p* in application *a* is allocated to SPM *s*. Similarly, If *Map[a, p, s]* = 0, Page *p* in application *a* is not mapped to SPM *s*. It's worth mentioning that main memory is considered as one of the SPMs with its own characteristics (access time). Total cost can be expressed as Equation (1):

$$Total_{Cost} = \sum_{a \in APP,\, p \in a.PAGE,\, s \in SPM} Map[a, p, s] \times Cost[a, s] \times Access[a, p] \quad (1)$$

The optimization problem can be formulated as Equation (2):

$$\min(Total_{Cost}) \quad (2)$$

Subject to:

1. Unique mapping

$$\forall a \in APP, \forall p \in a.PAGE: \sum_{s \in SPM} Map[a, p, s] = 1$$

2. Capacity constraints

$$\forall s \in SPM: \sum_{a \in APP,\, p \in a.PAGE} Map[a, p, s] \le Cap[s]$$

This optimization problem can be solved with Integer Linear Programming (ILP). Figure 9 shows the time required for obtaining the optimal mapping by ILP. Even for very small multi-core platforms, seconds are needed for computation and the required time increases exponentially with the number of SPMs (cores). For more than 49 SPMs, the ILP solver could not find any solution because of memory constraints.

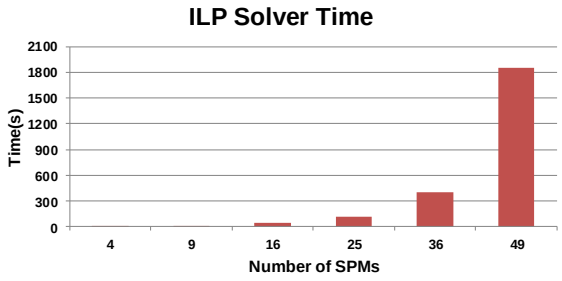

Figure 8. ILP solver execution time

Under the assumption of constant cost for each access to a page, regardless of its on-chip location, a single value can be given to each page. In this case, Knapsack Problem can be reduced to SPMPool Problem. Cost of accessing to each page can be varied based on proximity to its home core. Therefore, SPMPool has more options to explore than knapsack problem. As Knapsack Problem is NP-complete, solving SPMPool problem is not feasible in polynomial time.

Finding the optimal memory map is infeasible due to the size of search space. Instead, SPMPool uses heuristics to approximate the optimal mapping. This memory mapping cannot be generated continuously. There is overhead associated not only with determining the desired mapping, but also with moving memory on and off chip. In order to provide dynamic adaptability, the mapping does need to be regenerated periodically - for example, when the system state changes by an application starting or exiting.

5. SPMPOOL

5.1 Overview

In order to support the pooling of on-chip memory resources and make them available to any executing application as we have outlined, we need to augment the existing architecture with both hardware and software mechanisms. To provide each application a virtual private address space (Figure 9a) with data that may be physically distributed throughout the on-chip network, we must allow the applications to both address and access remote SPMs, and store information about the physical location of their data. The runtime manager must receive and store application specific information and subsequently make online memory mapping decisions using this information when triggered by an event. In this section, we specify these additions that enable SPMPool resource sharing.

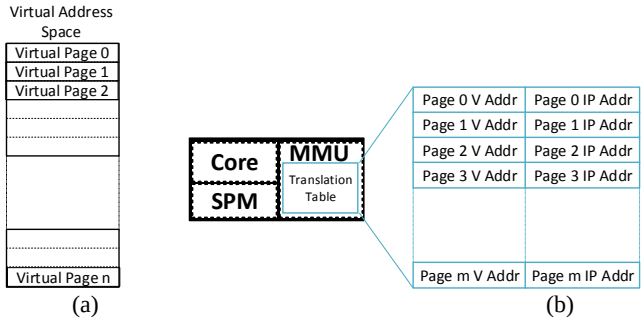

Figure 9. (a) SPMPool layout of virtual SPM address space and (b) translation table that contains SPMPool address translation information

5.2 Memory Access Mechanism

There are some additional hardware components and organizational specifications required for applications to access memory stored in distributed SPMs. Each tile needs an augmented translation table that contains SPMPool mapping information (Figure 9b). These SPMPool-specific translation tables specify virtual address to physical on-chip SPM address mappings for any of the application's pages that are stored on-chip. Any memory access initially checks this translation table for the desired page. If an entry for the page exists, a local or remote SPM access is initiated depending on the mapping indicated. For applications to be able to access data in remote SPMs, virtual memory management generates an intermediate physical address (IPA) for each SPM page. In this scheme, the SPM IPAs (held in the translation tables) can be decoded to determine where the specific SPM page resides physically on chip. The MMU on each tile contains hardware to handle direct SPM access requests to and from remote tiles. If an application's page is not in its local translation table, the memory access proceeds as a standard main memory access would. Figure 10 illustrates a sample remote SPM access by an executing application. The application executing on Core 0 initiates a memory load by sending the request to its local MMU, which contains the translation table. In this case, based on the virtual to IPA translation of the requested address, the MMU determines that the data resides on the remote SPM located on Tile 1. It sends a request over the NoC for the data, and that request is received and serviced directly by the MMU on Tile 1 (red path in Figure 10). The data is returned to the requesting core through the same path (green path in Figure 10).

Figure 10. Example remote SPM access

5.3 SPMPool Event

An SPMPool event is defined as a change in the workload that triggers the runtime manager to update the memory mapping. As previously mentioned, the event model consists of two types: application start and application stop. When either event occurs, there are some SPMPool-specific tasks to complete before the new memory map can be generated.

In the case of application start, the application is initially assigned randomly to a tile with an available compute resource by a designated central entity - the runtime manager. The manager also receives an incoming application's page list and stores all lists for executing applications. This SPMPool-specific information is generated at compile-time, and is discussed later in this section. Upon receiving a newly assigned application's page list, the manager generates the new mapping (see *generate_new_map()* below). Once the memory map has been generated, the appropriate pages are paged in from main memory and the manager sends updates to any remote tiles whose application's translation tables have changed as a result of the updated mapping.

In the case of application stop, the application's virtual pages are de-allocated from physical SPM pages and written back to main memory. Once the exiting application has vacated SPM space, the same sequence of steps is taken to generate an updated memory map.

5.4 SPMPool Components

This section describes in detail each step in both the runtime and compile-time components of SPMPool.

5.4.1 SPMPool Runtime Manager

The runtime component of SPMPool is invoked whenever triggered by an application event. Provided the list of pages and values for all active applications, as well as the type of triggering event, the runtime generates the virtual to physical memory mapping. Pseudocode for the SPMPool runtime component routine can be seen below.

```
wait_for_event (e)
     switch (e)
     case (start):
          new_map =
               generate_new_map(current_map + e.app.pages)
          memory_reconfigure()
     case (stop):
          new_map =
               generate_new_map(current_map - e.app.pages)
          memory_reconfigure()
```

Given the information provided by the compiler, the runtime manager generates a memory mapping based on the set of applications executing on the system by executing *generate_new_map(e.apps)*. The runtime manager stores information about each executing application's page list, all free SPM pages, and all pages on-chip. It uses this global state information to make memory mapping decisions. The specific policies for making decisions are defined in Section 5.5.

5.4.2 Compile-time Static Analysis

SPMPool needs to perform some preprocessing on application binaries to determine the working set of memory pages and the access pattern to those pages for each application. For every page in an application's working set, an ordering of accesses to the given page is created. This is done by scanning a memory trace of a sample execution for the given application. Using the page access information, a single value is generated for the page - in this paper, this value is simply the total number of times the associated page is accessed over the application's entire execution. The application's list of pages and associated values is then passed to the SPMPool runtime upon the application's entry into the system.

5.5 Memory Mapping Policies

Given the information provided by the compiler, the runtime manager generates a memory mapping based on the set of applications executing on the system by executing *generate_new_map(e.apps)*.

A key part of SPMPool is to generate the mapping between physical and virtual memory resources. As we showed earlier in the formal problem definition, obtaining an optimal mapping even for a single instance in time is unrealistically complex for many-cores. In this section some alternative best-effort strategies for generating memory mapping policies are discussed. These policies require the page access information for each executing application, as well as some knowledge about the current mapping. The placement mechanism is transparent to the application and developer - no other information is required, including code annotation for explicit SPM storage.

Each mapping policy attempts to prioritize the pages with higher need for memory resources and consists of two phases:

1. Deciding if a page should remain on-chip or go off-chip.
2. Mapping virtual pages to physical memory locations.

These two phases encompass the entire memory placement decision process. We will show in Section 8 that it can be completed in constant time complexity.

5.5.1 Most-Accessed Mapping

This placement policy attempts to maintain the most relevant pages for all executing applications in SPMs using the information extracted from the static analysis mentioned above. Based on the values assigned to each page that reflect the number of accesses to that page, all executing applications' pages are compared, and the most accessed overall are placed on-chip. In phase 1, all applications' working set of pages are combined and sorted based on their access value, resulting in a sorted page list of all pages in use by executing applications. We subsequently split this list into two sub-lists: the first N pages make up the on-chip list, where N is the total number of on-chip SPM pages; all remaining pages make up the off-chip list. In phase 2, the pages in the on-chip list are assigned to SPMs: each page in descending order of value is assigned to the free SPM page nearest its home core (the tile on which its owner application is executing). All remaining pages (the off-chip list) are placed in off-chip memory.

5.5.2 Neighborhood Mapping

This placement policy attempts to place an application's high-value pages in SPMs as close as possible to its home core by first

claiming all of the pages in its local SPM, and subsequently searching outward for free SPM space in the "neighborhood" of surround tiles. Phase 1 and 2 are combined in this policy because for each page in the incoming application's working set, we first attempt to place the page in the local SPM. Unlike the previous policy, this one gives applications priority allocation on local SPMs. If the entire local SPM consists of pages from the same application with higher access values, we then progressively increase the search space for remote SPM allocation without re-mapping any executing applications' pages on-chip. We search for a destination in groups of SPMs defined as having an equal hop distance from the home tile. If there is free space in an SPM, or if replacing the existing page would lower the overall access latency, the page is assigned. Once the search neighborhood is expanded past a threshold without an assignment, the pending page and all of the entering application's unmapped pages are relegated to off-chip memory.

5.5.3 Local-Only Mapping

In this mapping, each application can only use its local SPM. Therefore, all pages in an application are sorted based on their values and the top *S* pages are mapped to the local SPM while the other pages are mapped to off-chip memory. *S* is the number of available pages in each SPM. This emulates policies that do not support remote SPM utilization.

The following section compares these different memory mapping policies using trace-based simulations for diverse workloads on a number of different configurations.

6. EXPERIMENTAL SETUP AND RESULT

6.1 Experimental Setup

Our simulation infrastructure is designed for memory access energy and latency evaluation of user-specified workloads on a variety of chip configurations. We use Noxim [17] to obtain the NoC's cycle counts. CACTI [16] is used to obtain memory latency information. We assume 65nm SRAMs. Each tile is associated with a 64KB SPM. All virtual and physical memory is divided into 4KB pages. There are no caches present, and we assume all data and instructions are stored in RAM. Our simulation infrastructure is trace driven; an analyzer parses the traces and determines the value of each page in every application.

For our experiments, we used a variety of C benchmarks from the SPEC2006 suite in order to emulate highly varied high-performance workload behavior both between and within simulations. The benchmarks were not annotated for any SPM specific assignments. To evaluate the scalability of different memory mapping policies we simulated a range of NoC configurations from 4x4 to 32x32. Furthermore, each NoC configuration was simulated at multiple utilization points. Trace selection and entry and exit time was randomly generated independently for each simulation set so as to emphasize the unpredictability and variance of potential workloads.

Our goal is to show the efficacy of our method as the first run-time management solution for software controlled memory many-cores. In our experiments, the previously discussed placement policies are implemented and compared to each other.

6.2 Experimental Results

Figure 11 shows the improvement in overall memory access latency our best SPMPool placement policy achieves over the local-only policy for both 4x4 and 8x8 configurations at different core utilizations. As the number of utilized cores increases, the advantage of SPMPool over non-sharing organizations decreases. This is expected: one substantial benefit of SPMPool is it provides active memory-intensive applications access to SPM resources on remote tiles that otherwise would remain idle and unused. As the number of active tiles and applications increases, the competition for on-chip memory resources between memory-hungry applications negates the idle-tile advantage. However, we still observe ~7% reduction in overall memory latency when the 8x8 configuration is 100% utilized. This edge is from the fact that we compare page values between applications to allocate physical memory resources to pages that yield overall benefit.

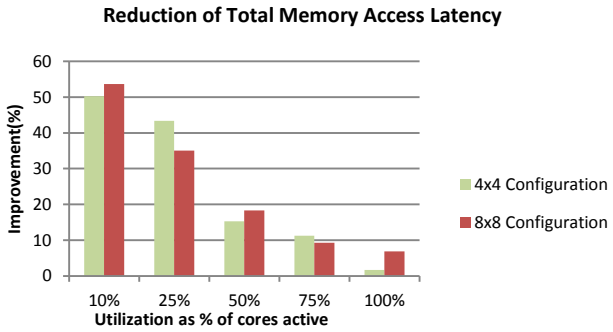

Figure 11. Percentage of SPMPool latency improvement with respect to Local method

From the results in Figure 11 we also confirm the workload dependence of SPMPool. Every simulation shown is run with a unique randomly generated workload - this is why we don't necessarily observe a consistent relationship between configurations through the different utilizations.

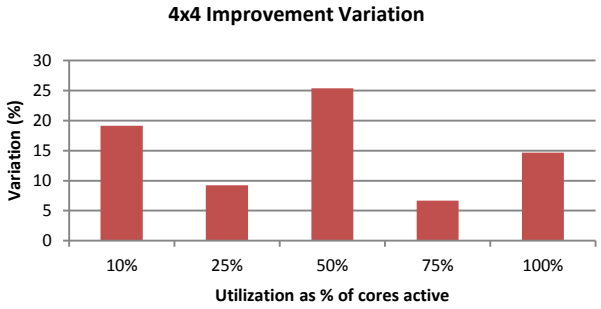

Figure 12. Maximum improvement Variation (%) between sets of applications for 4x4 configurations

The total energy consumption for each test case was also calculated. Due to our energy consumption model being closely tied to our access latency model, the relative improvement for energy consumption was very similar to the latency improvement in every case.

SPMPool performance is highly related to the set of applications and their scheduling. In our experiments we observed a significant variation (up to 25%) between memory access latency improvements for different sets of applications. Figure 12 shows the maximum latency improvement variation for different sets of applications.

It is also important to address the dependence of SPMPool upon the overall application mapping policy. For all of our experiments, we assign incoming applications to tiles randomly. As an alternative, we did some simulations that instead found a pseudo-optimal tile to assign each application based on the memory map when the application entered the system. We observed, in the best case, less than 10% difference in the overall memory latency of a simulation between random application mapping and our smart placement across all utilization points. We do not consider this advantage significant enough to justify a complex application mapping policy, therefore all simulations presented implement random application mapping.

Figures 13, 14, and 15 include a comprehensive comparison of all SPM mapping policies covered in this paper, for all utilizations on configurations from 4x4 to 16x16. The overall memory access latency is reduced in all cases. One additional placement policy not discussed previously that we included is a simplified version of the most-accessed policy. The most-accessed policy re-maps all on-chip memory pages every time it generates a memory map, which requires all executing applications to halt and can induce excess memory migration overhead that is discussed in-depth in the next section. To reduce the overhead, we modified the policy to only map application pages that are migrating from off chip to on, limiting memory migration to between on- and off-chip memory only, not between SPMs. This simplified version of most-accessed still outperformed the local and neighborhood policies in every case.

The range of utilizations simulated for 4x4 and 8x8 consisted of five points from 10% to 100%. The 16x16 configuration was run for simulations from 12% to 75% utilization (Figure 15). We believe that as NoC size reaches hundreds of cores, due to the limits on software degree of parallelism and the power envelope, 100% utilization is not a common use case. Additionally, we completed simulations for a 32x32 configuration at two different low-utilization points that yielded results almost identical to the 16x16 configuration (Figure 15). As expected with so much on-chip memory resources available to share, the advantage of any SPMPool memory placement policy over local increases with the configuration size (Figure 16).

Relative performance between SPMPool memory placement policies changes as configuration size grows to hundreds of cores (Figures 15, 16). This makes sense - as configuration size grows the severity of across-chip remote SPM access penalty increases. Therefore, growing neighborhoods of an application's pages around its home core yields a better result than allowing its pages to potentially become sparsely distributed throughout the chip over the course of execution. However, as the number of cores reaches 1024, the scalability of this version of SPMPool becomes an issue – an on-chip memory access may be more costly than an off-chip access in the most extreme cases. For 32x32 experiments, we forced a page to be placed in off-chip memory if the access latency from the page's home core to its assigned SPM exceeded a threshold. Multiple pools become necessary due to severe delay accessing remote tiles across chip. A single centralized runtime manager is no longer sufficient - we must move to a distributed management topology. These are topics of ongoing work to extend SPMPool as a complete scalable solution.

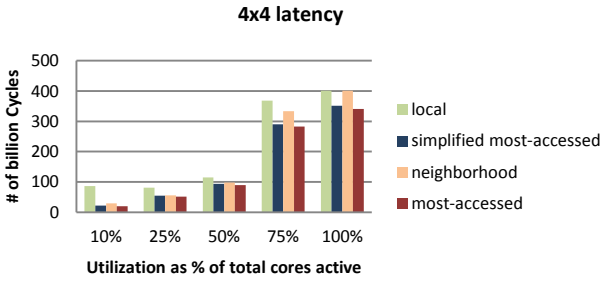

Figure 13. Total memory access latency for 4x4

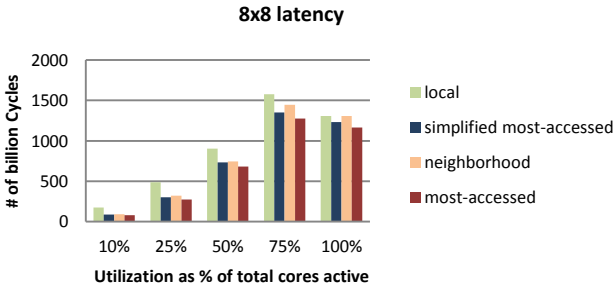

Figure 14. Total memory access latency for 8x8

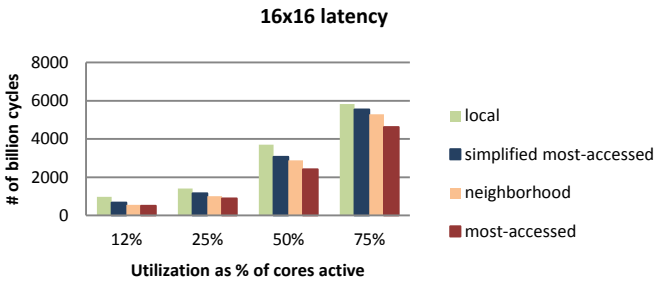

Figure 15. Total memory access latency of 16x16 configurations for different workloads

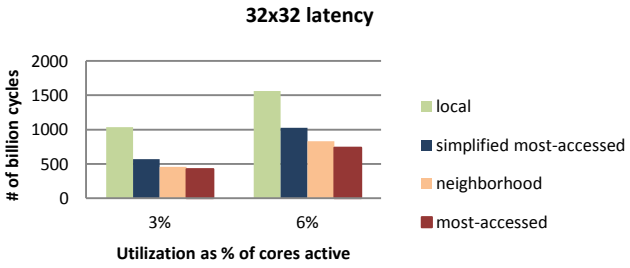

Figure 16. Total memory access latency of 32x32 for different workloads

7. TEMPORAL VARIANCE OF MEMORY USAGE

In each application, the frequency at which different pages are accessed varies over the course of execution; some pages might be accessed predominantly shortly after the start of execution, while other pages are primarily referenced in a later phase of execution. Ignoring this phasic behavior of memory accesses and mapping an application's memory a single time (at the start of execution) can result in severe performance penalty due to excessive accesses to off-chip memory. This characteristic and the inherent opportunity it presents for a more efficient memory mapping is illustrated by Figure 6 and the related example in Section 2.

To address this issue, we propose to define different phases for any application and perform memory re-mapping at the start of each phase. Therefore, the runtime manager will be triggered with a new type of event in addition to the stop and start of applications, which is phase change. Shorter phases will impose more overhead due to more frequent memory map updates, while potentially reducing the memory access latency. In this work, we illustrate the benefit of this re-mapping by manually fixing the number of phases for each application.

We repeated the experiments of Section 6.2, considering five phases evenly dividing each application. Figure 16 and Figure 17 show the memory access latency improvement compared to the single-phase application model we used previously, in 4x4 and 8x8 system configurations.

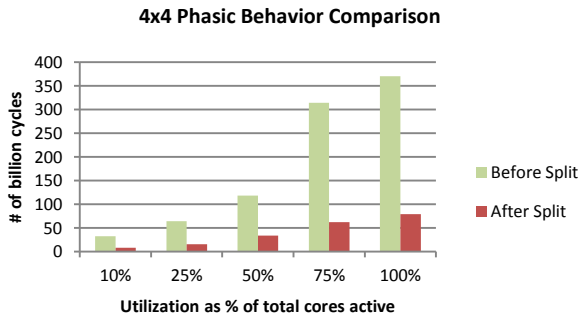

Figure 17. Memory access latency, before and after splitting application to 5 different phases, for 4x4 configurations

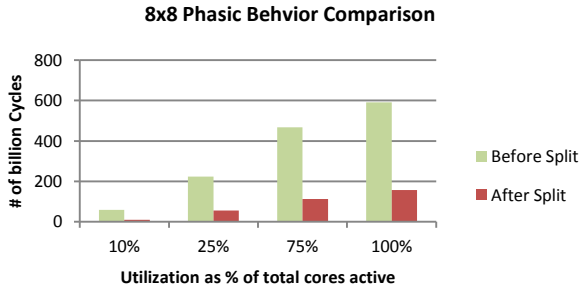

Figure 18. Memory access latency, before and after splitting applications to 5 different phases, for 8x8 configurations

These results show significant improvement (80%) even for a naïve selection of application phases. Considering 10 phases for each application, experiments yield slightly better results over the 5 phase organization of the same workload. More thorough investigation of defining memory access phases proves to be promising future work.

8. Application Mapping

Relative assignment of applications to cores in NUMA many-cores impacts the memory access latency of the system. For all experiments discussed to this point, we assumed each application upon dispatch is randomly mapped to an available core without any knowledge of current programs executing. To quantify the degree of impact application placement has on a memory placement technique such as SPMPool, we re-evaluated a subset of the previous experiments while employing alternative application mapping strategies. The application placement strategy we choose to compare with the random application placement searches for a chip region with memories that are accessed least by existing cores. The SPMs in this region potentially have the minimum amount of conflicts with other cores in terms of memory accesses, and contain the least critical data on-chip for existing applications. As shown in Figures 19 and 20, the impact that using an intelligent task placement policy, such as our minimum conflict policy, has on overall memory access latency is negligible for any memory placement strategy discussed in this paper.

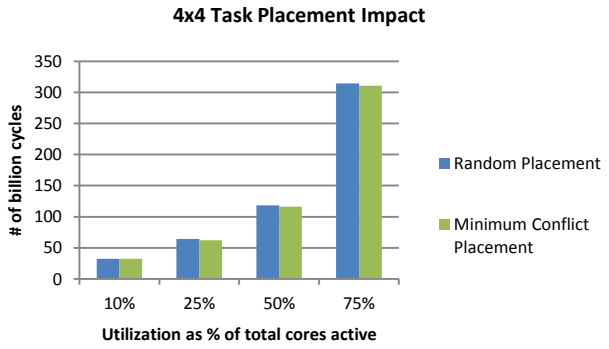

Figure 19. Memory access latency for the same workload on a 4x4 configuration using different task placement policies

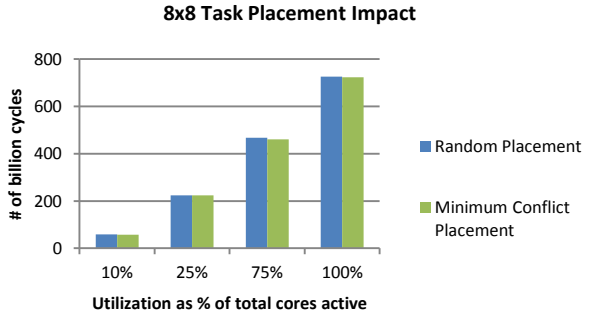

Figure 20. Memory access latency for the same workload on a 8x8 configuration using different task placement policies

9. OVERHEAD

Using SPMPool imposes some overhead on the system. This overhead includes communication, computation, storage, and memory re-mapping overhead. In this section, the different overhead components of SPMPool are investigated. Our analyses and results show that SPMPool has a relatively low overhead.

9.1 Communication Overhead

The runtime manager needs to communicate with cores when generating a new memory map. Three events trigger this communication: start, stop, and phase change of any application. All the communication has constant time complexity for a fixed configuration size.

9.1.1 Application Start

The list of memory pages in the application along with their number of accesses should be sent to manager on application entrance. Assuming *N* is the number of pages that can be held in all SPMs, information of *N* pages in the application with highest number of accesses needs to be sent to the manager. After determining the new map, the manager sends the information to update the translation table in each tile if the memory mapping associated with the application at that tile has changed. The manager needs to send at most *N* messages for this purpose. Therefore, the total communication overhead consists of sending 2**N* messages (which has constant time complexity based on configuration size). After creation of a new map, we should change the data placement. The overhead associated with the change in data placement will be covered in Section 8.3.

9.1.2 Application Stop

Whenever one application stops (completes its execution), we no longer need to keep its data in SPMs. The application sends a message to the manager to indicate its completion. The manager generates a new memory map after removing the application's pages from on-chip memories. After determining the new map, the manager sends the information to update the lookup table in each core if necessary. Assuming *N* is the number of pages that can be held in all SPMs, at most *N* messages need to be sent, which has constant time complexity based on configuration size. Consequently, data placement should be changed.

9.1.3 Application Phase Change

Any change in application phase can be considered as an application exit with a consequent immediate application entrance (no need for data movement between stop and start).

9.2 Computation Overhead of Mapping Algorithm

After any event, the manager will run the memory mapping algorithm to obtain the new map. We consider the SPMPool mapping policy in Section 5.5.1 (most-accessed) for this part, but the overhead of other mapping algorithms can be obtained in the same way. Assuming the manager has access to the information for each application (host core and number of accesses, in descending order, for each page), the mapping policy is implemented in two parts:

1. Decide if a page should remain on-chip or go off-chip. Time complexity of completing this phase is O(1). If we have the first *N* pages sorted (*N = #SPMs * #pages_per_SPM*) based on their number of accesses, we can sort the list of other pages (which should go off-chip) later. The manager runs this algorithm on occurrence of each SPMPool event. It already has the sorted number of accesses of all other pages in other applications combined. The manager also has the sorted number of accesses for pages in application *A*. Therefore, it can find the first *N* pages with highest access value using a simple merge operation.

2. Map virtual pages to physical memory locations. Time complexity of this phase is O(*N * #SPMS*) = O(1), where *N* is the number of pages in the on-chip list *(#SPMS * #pages_per_SPM*). For each *N* page, the manager keeps the list of closest to farthest SPMs from the host tile of the associated application. The manager traverses this list and assigns the page to the first SPM with an empty slot. In the worst case, for each on-chip page we should look at all SPMs to find a free slot.

By adding time complexity of those two phases of the algorithm, we determine SPMPool runs the mapping algorithm in constant time. All overhead incurred on the runtime manager to this point does not require any applications to suspend execution and therefore has a low impact on the overhead of the entire system.

9.3 Storage Overhead

For maintaining the virtual to physical memory mapping and also fast execution of the mapping algorithm, the manager and each tile need to hold some SPMPool-specific information. If the total capacity of SPMPool is *N* pages, the manager needs to keep the number of accesses for *#apps * N* pages. To locate its data in physical memory, each tile needs to have a translation table for any on-chip pages belonging to the application executing on its core. This table is also necessary for non-SPMPool organizations.

9.4 Memory Migration Overhead

After generating a new memory map, some application pages may need to be moved between off-chip and on-chip memory, as well as between SPMs. During this migration, affected applications should halt their execution. SPMPool, in the most extreme case, may change the entire memory mapping and thus all pages residing in the pool of SPMs need to be migrated. The probability of this occurring, however, is low. Even in non-SPMPool organizations, some degree of memory migration is inevitable. Due to the fact that SPMs are typically small (64KB in this work), and interconnection networks are relatively high bandwidth, overhead due to memory migration is low.

We monitored the memory migration rate in the experiments described in Section 6.2. Although Intel's SSC [18] provides up to 64GB/s bandwidth for on-chip communication, we conservatively calculated the overhead assuming a bandwidth of 8GB/s for on-chip communication and 160 MB/s for accessing main memory. We compared the overall memory migration time to the total simulated execution time to estimate the memory migration overhead. Our results, depicted in Figure 18, show that for 4x4 and 8x8 configurations the memory migration overhead, for most cases, is in the range of 2% to 6% considering the most-accessed policy. Overhead for the 8x8 configuration is slightly higher than 4x4 configuration. This trend continues in systems with configuration sizes of 16x16 and 32x32. The overhead increases with system configuration size due to longer communication paths and more memory migrations.

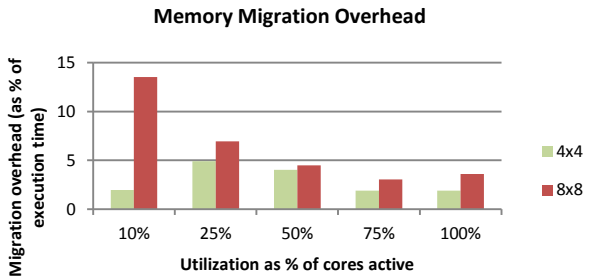

Figure 21. Memory migration overhead as a percentage of total workload execution time

If we account for all four types of overhead we outlined (Sections 8.1 - 8.4), SPMPool is a relatively small burden in terms of time and memory overhead to the whole system. This encourages the exploitation of temporal variance in applications in order to perform more frequent memory re-mapping and maintain a low overhead.

10. RELATED WORK

SPM mapping techniques have been previously investigated for both uni- and multi-core execution environments. Typically SPM-based systems load data into on-chip memory with explicit instructions inserted into application code by the compiler [11,14]. The instructions can be generated by API supported user annotations in the program, as well as through static analysis of the program by the compiler itself. There has been extensive work done previously on compiler based static analysis techniques for mapping data onto local SPMs in uni-core processors [1,2,3,4,11,12,13,14,15]. [1] proposes a compiler extension that partitions array data and allocates the most used segments to a scratchpad. However, numerous solutions of this kind do not consider the spatial or temporal sharing of the SPM with other applications at runtime. To support contention of a single SPM's resources by concurrently executing applications, [2,4] propose static analysis techniques. These static analysis techniques assume that the run-time working sets can be predicted accurately at design time. SPMPool's runtime manager allows it to adapt to dynamic workloads, mitigating this limitation.

More recently in the multi-core domain, Bathen et. al. propose to allocate data at runtime based on fine-grained user-specified priorities for multiple applications that compete for local SPM space [5]. [9] has considered a system in which shared scratchpad resources are distributed throughout multi-core processors. Although they do use a NUMA model, they still suffer the same adaptability consequences that all static solutions do. In [6], a multi-phase program analysis is proposed to partition arrays by simulating and monitoring the program's memory accesses, and based on the simulation input the program is compiled again and the SPM allocation algorithm is implemented. This technique exploits the simulation feedback, but cannot predict the runtime workload and the constraints other applications may put on it. SPMPool uses compile-time information to dictate its runtime memory mapping decisions in order to resolve spatial contention between concurrently executing applications.

In contrast to previous work discussed, we consider sharing distributed on-chip SPMs of many-core processors among applications throughout the chip. State of the art approaches for on-chip distributed memory for caches, e.g. [7], have similar sentiments for sharing cache resources on multi-core. Lee et. al. propose to share caches on a many-core chip [8]. However, no runtime solutions for sharing SPM's in multi- or many-core processors exist, and the cache-based solutions have limited flexibility due to hardware control. The static analysis solutions for SPM's have been limited to multi-core at most. These solutions are insufficient for a chip with hundreds of cores executing numerous applications all sharing the entire on-chip memory pool.

Some emerging architectures do address the need to adaptively share resources for future many-core execution environments. The invasive computing paradigm expresses a similar sentiment to ours through adaptive resource-aware programming, but its focus is on processing elements [10]. The SPMPool approach could complement a compute-focused paradigm like invasive computing.

11. CONCLUSION AND FUTURE WORK

As the initial investigators in providing dynamic runtime management for distributed scratchpads in many-cores, we proposed SPMPool: a viable solution for sharing SPMs in an environment consisting of unpredictable workloads. Our results and analysis show that the introduction of SPMPool can achieve up to 55% reduction in overall memory access latency for underutilized systems with reasonable overhead. This reduction can be significantly increased by accounting for the different phases of memory access patterns typical applications experience.

Because this work is introductory, we've identified multiple promising avenues for future work throughout the paper. If we leverage the knowledge we have about the phasic behavior of memory accesses to dictate when we perform memory re-mapping, we can reduce memory access latency even further. In all cases, we initially targeted total memory access latency as the primary performance metric, but it is important to quantify the impact of SPMPool on energy consumption as it is applied to embedded environments. It is also important to account for how SPMPool applies as chips scale to 1000s of cores. At a certain point, it is no longer beneficial to maintain a single pool for the whole system. Instead, we can divide the chip into multiple pools with distributed management.

12. REFERENCES

[1] M. Verma, S. Steinke, and P. Marwedel. Data partitioning for maximal scratchpad usage. In *ASP-DAC*, 2003.

[2] V. Suhendra, A. Roychoudhury, and T. Mitra. Scratchpad allocation for concurrent embedded software. In *CODES+ISSS*, 2008.

[3] V. Suhendra, C. Raghavan, and T. Mitra. Integrated scratchpad memory optimization and task scheduling for MPSoC architectures. In *CASES*, 2006.

[4] H. Takase, H. Tomiyama, and H. Takada. Partitioning and allocation of scratch-pad memory for priority-based preemptive multi-task systems. In *DATE*, 2010.

[5] L. Bathen, N. Dutt, D. Shin, and S. Lim. SPMVisor: dynamic scratchpad memory virtualization for secure, low power, and high performance distributed on-chip memories. In *CODES+ISSS*, 2011.

[6] A. Marongiu and L. Benini. An OpenMP compiler for efficient use of distributed scratchpad memory in MPSoCs. *TC*, 61(2), 2012.

[7] A. Sharifi, S. Srikantaiah, M. Kandemir, and M. Irwin. Courteous cache sharing: Being nice to others in capacity management. In *DAC*, 2012.

[8] H. Lee, S. Cho, and B. Childers. CloudCache: Expanding and shrinking private caches. In *HPCA*, 2011.

[9] M. Kandemir, J. Ramanujam, and A. Choudhary. Exploiting shared scratch pad memory space in embedded multiprocessor systems. In *DAC*, 2002.

[10] J. Henkel, V. Narayanan, S. Parameswaran, and J. Teich. Run-Time Adaption for Highly-Complex Multi-Core Systems. In *CODES+ISSS*, 2013.

[11] S. Udayakumran, A. Dominguez, and R. Barua. Dynamic Allocation for Scratch-Pad Memory Using Compile-Time Decisions. *TECS*, 2006.

[12] A. Pabalkar, A. Shrivastava, A. Kannan, and J. Lee. SDRM: Simultaneous Determination of Regions and Function-to-Region Mapping. In *HiPC*, 2008.

[13] S. Steinke, L. Wehmeyer, B. Lee, and P. Marwedel. Assigning Program and Data Objects to Scratchpad for Energy Reduction. In *DATE*, 2002.

[14] P. Francesco, P. Marchal, D. Atienza, L. Benini, F. Catthoor, and J. Mendias. An integrated hardware/software approach for run-time scratchpad management. In *DAC*, 2004.

[15] D. Cho, S. Pasricha, I. Issenin, N. Dutt, M. Ahn, and Y. Paek. Adaptive Scratch Pad Memory Management for Dynamic Behavior of Multimedia Applications. *TCAD*, 28(4), 2009.

[16] N. Muralimanohar et al. CACTI 6.5. HP Laboratories. 2009.

[17] F. Fazzino, M. Palesi, D. Patti - URL: http://sourceforge. net/projects/noxim, 2008.

[18] J. Howard, S. Dighe, Y. Hoskote, et al. A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS. In *ISSCC*, 2010.