Runtime memory allocation in a heterogeneous reconfigurable platform

Vlad-Mihai Sima, Koen Bertels
Computer Engineering
Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands

Abstract—**In this paper, we present a runtime memory allocation algorithm, that aims to substantially reduce the overhead caused by shared-memory accesses by allocating memory directly in the local scratch pad memories. We target a heterogeneous platform, with a complex memory hierarchy. Using special instrumentation, we determine what memory areas are used in functions that could run on different processing elements, like, for example a reconfigurable logic array. Based on profile information, the programmer annotates some functions as candidates for accelerated execution. Then, an algorithm decides the best allocation, taking into account the various processing elements and special scratch pad memories of the the heterogeneous platform. Tests are performed on our prototype platform, a Virtex ML410 with Linux operating system, containing a PowerPC processor and a Xilinx FPGA, implementing the MOLEN programming paradigm. We test the algorithm using both state of the art H.264 video encoder as well as other synthetic applications. The performance improvement for the H.264 application is 14% compared to the software only version while the overhead is less than 1% of the application execution time. This improvement is the optimal improvement that can be obtained by optimizing the memory allocation. For the synthetic applications the results are within 5% of the optimum.** [1]

I. Introduction

As the complexity of applications increases, hardware and software engineers have to develop new approaches to provide the best possible performance. One solution is to use heterogeneous systems, which can include various processing elements like reconfigurable processing elements and DSP processor. In this way the flexibility of a GPP (general purpose processor) is combined with the speed of the other processing elements, while avoiding the costs and resources involved in developing custom ASIC-s. But with greater flexibility comes the problem of mapping the application efficiently, especially when the system is a dynamic system with multiple applications and an operating system.

We analysed a complex application, the H.264 video encoder, and determined that one important problem is the memory access overheads. The solution is to allocate the memory directly in the **scratch pad memory** (SPM) of each processing element. In this paper, we propose an on-line algorithm that manages the memory allocation called **AMMA** (adaptive memory mapping algorithm). The algorithm uses instrumentation to track memory allocations and annotated function execution, without any need for manual application modification. The programmer's only responsibility is to annotate the functions for which there are available implementations for other processing elements. Using this information the algorithm can decide which is the best memory allocation.

The paper is organized as follows: in Section II we briefly present the MOLEN programming paradigm for reconfigurable architectures and related work. Next, we give a motivational example and the problem definition. Our proposed solution to the problems exposed are presented in Section V. An extension of the AMMA algorithm is in Section VI. The results of the algorithm are shown in Section VII. In Section VIII, we present conclusions and outline new research directions.

[1]This research has been funded by the hArtes project EU-IST-035143, the Morpheus project EU-IST-027342 and the Rcosy Progress project DES-6392

II. Background and related work

The **MOLEN programming paradigm** [1] is a paradigm that offers a model of interaction between the components of a heterogeneous multi-core system. Using a 'one time' architectural instruction set extension the MOLEN programming paradigm allows for a virtually infinite number of new hardware operations to be executed on the reconfigurable hardware or any other processing element that supports this paradigm.

Previous work related to SPM-s, considers usually just one processing element, like [2] which gives an ILP formulation to the distributed stack allocation problem, using a complete trace of memory accesses. In [3] is presented a static method to determine the memory bank where a variable should be placed, based on the number of accesses and conflicts with others variables. Both approaches rely on detailed information, which can be hard to obtain.

In [4] an algorithm is presented that determines the program points where local (stack) variables have to be transferred to/from RAM to the SPM. In contrast to previous work, [5] instruments just the dynamic memory management primitives along with all the accesses to memory. A data identification algorithm is proposed and based on the results, DMA optimizations are implemented. In contrast with those two approaches, we target both local and dynamic variables and do not rely on static information.

At the boundary between static and dynamic allocation, [6] performs a load time optimization that places the stack data in one of the memories using information computed at compile time, but taking into account the size of the memory at runtime. Compared to this approach, we manage both heap and stack data.

In [7] a dynamic SPM stack manager is presented. The main advantages are that it doesn't need profile information, and the size of the SPM only needs be known at runtime. The manager optimizes the transfers between main memory and SPM in order to reduce power consumption. Providing both heap and stack management, [8] describes an iterative algorithm that determines the allocation of stack, global and heap variables and introduces transfer code when necessary. Our approach tries to allocate memory directly where it is more efficient, without the need of transfers.

In [9] a scratch pad memory management is presented which

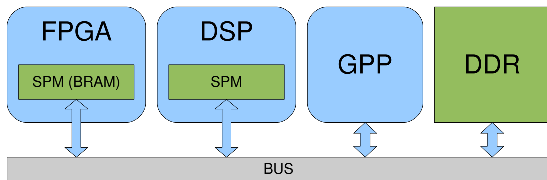

Fig. 1: Hartes platform architecture

```
#pragma call_hw FPGA
int satd(char *p1,char *p2,
  int s1, int s2);
```

Fig. 2: Annotated function

```
memcpy(spm_buf1,buf1,size_buf1);
memcpy(spm_buf2,buf2,size_buf2);
r = satd(spm_buf1,spm_buf2,4,4);
```

Fig. 3: Code with transfer added to move data between main memory and SPM

relies on compile time information to decide how to divide the scratch pad memory between multiple applications. The algorithm can extract at compile time information about a specific loop type and then, at runtime, give each application memory, based on the amount requested, but compared to our approach it does not take into account the possible performance improvement obtained by the allocation of each memory block.

There are also hardware solutions, like those proposed in [10] and [11], which modify the MMU of the processor in order to manage the scratch pad memory. These solutions are complex as they require extensive hardware changes, and usually can't be applied to existing platforms.

III. Motivation - H.264 application

We started by trying to map the open source implementation of the H.264 video codec on a heterogeneous platform. The H.264 video codec is a state of the art codec in video compression. The heterogeneous platform is the HARTES [12] platform which contains an ARM as general purpose processor, a Xilinx Virtex4 FPGA and an Atmel Magic DSP. The general architecture is described in Figure 1. We focus our analysis on the encoder part of the codec. By profiling the application two **kernels** (computationally intensive functions) were identified: satd (around 20% of the execution time) and sad (around 10% of the execution time). FPGA implementations were generated for those functions and MOLEN pragma annotations where introduced in the application source code. An example of such an annotation is given in Figure 2. One problem is that the FPGA kernels can access just the BRAM from the FPGA, which in fact constitutes the local SPM. To take advantage of the acceleration of the FPGA kernels, we had to modify the code and insert a memory transfer before each annotated function invocation, as in Figure 3. As the access to the memory was not linear, the memory accesses introduced unnecessary transfers, which resulted in an application slowdown of more than 2x.

Our proposed solution is to allocate the data directly in the SPM thereby reducing substantially the need for DDR based memory accesses. This implies detecting all the places where memory is allocated, and most important, as the size of the SPM is small, determine which of the allocations are worth

```
buf1 = malloc_spm(size_buf1);
buf2 = malloc_spm(size_buf2);
...........................
r = satd(buf1,buf2,4,4);
```

Fig. 4: Code modified so that buf1 and buf2 are allocated directly in the SPM

to allocate in the SPM. For example in Figure 4 both buffers are allocated using a special allocation function, so they will always reside in the SPM. Also the SPM can be used by multiple applications, so the available size can vary from one execution to another.

In summary, any algorithm that wants to solve those issues need to address the following:

- Problem 1: identify which allocations are candidates for allocation in SPM.
- Problem 2: determine the total needed memory size (which is especially challenging for dynamic memory allocation).
- Problem 3: prioritize between various allocations in multiple applications based on the speedup.

We propose a dynamic allocation algorithm, that can track the memory allocations at application run time and then allocate the memory efficiently, by inserting code similar with the one in Figure 4. This is done for the allocations that will bring the highest overall improvement in performance. This improvement will be determined based on the function speedup and on the execution count.

IV. Problem definition

The problem can be split in two parts: the allocation tracking and the mapping algorithm.

A. Allocation tracking

Given the application source code, and the annotated functions, a list will be maintained with all the memory allocations that are used in those functions. For each combination of allocations used in a kernel we have an associated gain which represents how much time would be saved if that kernel would be executed accelerated. This gain can be obtained either at compile time, by profiling the software and the FPGA implementation, or can be computed at runtime by profiling the software and the FPGA implementations. We assume in the rest of the paper that we have this information provided at compile time. The gain is computed by multiplying the time saved by using the accelerated implementation with the execution count.

B. Memory mapping

The second part of the problem is to find which set of memory allocations have to be placed in the SPM to obtain the best performance. We assume that all the kernels execute sequentially. If parallel execution would be considered, the general idea of the algorithm holds, but some of the equation should be modified. Graphically, this is depicted in Figure 5 where the circles represent one kernel call, the rectangles represent the memory blocks and the big squares represent the memory. The gain for a kernel invocation can be obtained just if all the memory blocks are allocated to the SPM. In our example just $K1$ and $K2$ will run in hardware, while $K3$ will run on the GPP. With this mapping the gain would be of 400ms.

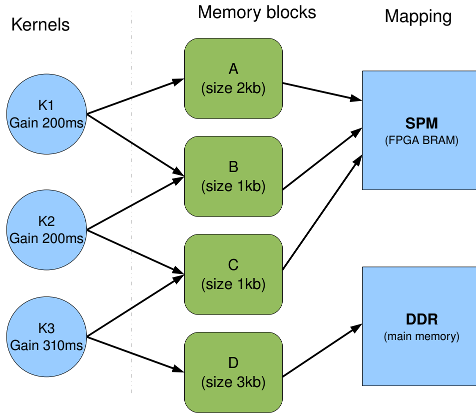

Fig. 5: Motivational example

V. AMMA framework

To solve the problems described, we present a dynamic framework, composed of three modules: the allocation tracking module, the execution module and the mapping algorithm. The application is **instrumented** with these modules by special optimization passes from the compiler. The allocation module tracks at run time all the allocations by managing a list with the starting and ending address of the currently allocated blocks (stack, global and heap). It also allocates the memory based on the decision of the allocation algorithm (scratch pad or main memory). The execution module, determines at runtime, for each of the annotated functions, if the data is allocated in SPM, and in that case uses the accelerated implementation. It also updates the gains associated with each combination of memory allocations, based on the known speedup of the current annotated function invocation. We call the lists of memory allocations, together with the associated gains the **memory allocation lists** or in short **MAL**. The MAL will be created at the first application execution, and the allocation module will update it, save it and restore it across multiple application runs.

The algorithm that computes the best allocation will be executed just at specific points in the program execution. It will be executed first after the MAL was generated for (at least) one application execution. Then, the algorithm must be called again in at least two cases. The first case is when the size of the available scratch pad memory changes because the operating system needs more memory for other applications. The second case is when an allocation is no longer optimal because of a change in the input data.

Using this approach, the only application modification needed is the annotation of the functions that can be run accelerated. Our approach is semi automatic and does not involve a manual analysis to determine exactly what allocations are needed by which kernels (and thus solves Problem 1). The size of the allocated memory will be determined just at runtime, giving more flexibility to the system (and thus solves Problem 2). If multiple applications are present the allocation can be done in such a way that the overall system performance is improved (solves Problem 3).

```
function:
  sp = cm_stack_allocate(id,size);
  cm_stack_add(id,sp,sp+size);
  ...
  sp = cm_stack_deallocate(id);
  return;
```

Fig. 6: Code added for functions for stack instrumentation (architecture dependent)

A. Allocation tracking module

The allocation module has two tasks: to manage the allocated memory blocks and, for new allocations, place them into the memory the allocation algorithm determined it would be best for the overall performance. Depending on the type of memory object, different mechanisms are used.

Global variables

All the global arrays are transformed to pointers and are initialized in a special initialization function *cm_global_init*. This is done so that the array addresses are not fixed at link time, but rather initialized at the application loading. This mechanism is one of the mechanisms used by the compiler when generating position independent code. In this function, there are two steps: the ranges of the arrays are added to the MAL for further checking and based on the allocation decided by the algorithm, each of the arrays is allocated either in main memory or in the scratch pad memory.

Stack variables

As the stack management is architecture dependent, so is the allocation module that treats stack variables. It has to be designed for each GPP the AMMA has to support.

Tracking the stacks for all functions would introduce unnecessary overhead. Instead, we performed an **optimization** by instrumenting just those functions which are found on a path from the root of the call graph (*main* function) to one of the annotated functions. If function pointers are used, we will always consider that such a pointer can point to an annotated function and instrument the function using the function pointers. Another **optimization** is to detect if any of the local variables in a function are used as parameters for other calls. If this is not the case, it means the local variables will never be used by annotated functions, so the instrumentation would be useless.

For each instrumented function a special header is added to the assembly code. First *cm_allocate* returns the address for the stack of the function, based on the decision that was taken by the algorithm. Then *cm_stack_add* is called which, based on an identifier, will add the starting and ending address to a list in the MAL. A pseudo-code of the wrapper is given in Figure 6, where *sp* is the stack pointer for local variables.

Heap memory allocations

To track the dynamic memory allocations, a wrapper around standard allocation functions is provided. The wrapper has two functions: add the allocations to the MAL and allocate the memory according to the decision of the allocation algorithm. One allocation is identified in this case by the address from which *malloc* was called.

B. Execution module

The execution module is used only for the functions annotated with a pragma by the programmer. It's role is to update internal data structures that track what memory

```
a[m] array of memory allocations
m[n] array of combinations of allocations

compute_scores(a,m);
sort(a,descending)

alloc = 0
for i = 0 to n
  if(alloc + a[i].size < memory size)
    mark a[i] to be placed in spm
    alloc + = a[i].size
```

Fig. 7: AMMA algorithm

areas are used by accelerated functions. Then, if all the data is placed in the appropriate memory, it will invoke the accelerated implementation.

C. Mapping Algorithm

The data available to the algorithms can be formally represented by:

- the set of memory allocations, $A = \{a_j, j = \overline{1..m}\}$, where m is the number of allocations tracked, and a_j is one allocated block.
- the combinations in which the allocations are used by each kernel. There can be multiple combinations used by each kernel, and one allocation can be in multiple combinations. Let the set of all combinations be $S = \{C_i, i = \overline{1..n} / C_i = (\{a_j\}, k_{i_{gain}})\}$. We denote by C_i a combination of memory allocations used in at least one kernel invocation and by a_j a specific memory allocation. The time gained by allocating all the memory in set C_i to scratch pad memory and running the associated kernel accelerated is denoted by $k_{i_{gain}}$.
- the size of the available scratch pad memory.

The idea of the algorithm is to order the memory allocations based on the memory score then allocate them to scratch pad memory, until the memory is full. The score is computed as the sum of the gains of the kernels that use that specific allocation. This is represented for memory location a_j as $a_{j_{score}}$ and we compute it using Equation 1. This is a very low overhead algorithm, as it needs just one sort operation on the memory objects and one pass to fill the memory. The pseudo-code is given in Figure 7.

$$a_{j_{score}} = \sum_{a_j \in C_i} k_{i_{gain}} \quad (1)$$

The main drawback of AMMA is that it does not take into account the fact that in order to obtain the gain, a complete group of memory allocations must be in SPM. For example, if we consider Figure 5, *K2* can't execute in HW unless **both** memory block *A* and *B* are allocated to SPM.

VI. AMMA extension

We now describe an extended version of the same algorithm, which we call AMMAe, which is more complex but provides better solutions than AMMA. We also give the ILP formulation of the problem, which will be used as benchmark as it guarantees to compute an optimal solution.

AMMAe algorithm

We now discuss an extension to the proposed AMMA algorithm where 2 new elements are introduced. We first present the new equations and then discuss their meaning. We introduce the following modified equations to compute the scores for each memory object:

```
a[m] array of memory allocations
m[n] array of combinations of allocations

alloc = 0
do
  changed = false
  compute_scores_extended(a,m)
  sort(a,descending)
  for i = 0 to n
    if(alloc + a[i].size < memory size)
      mark a[i] to be placed in spm
      alloc + = a[i].size
      remove a[i] from a
      changed = true
      break
while (not changed)
```

Fig. 8: AMMAe algorithm

$$s_i = \sum_{a_j \in C_i, a_j \text{ is not allocated}} a_{j_{size}} \quad (2)$$

$$a_{j,i_{score}} = \begin{cases} 0 & \text{, if } s_i < f \text{ or } a_j \notin C_i \\ k_{i_{gain}} \cdot \frac{a_{j_{size}}}{s_i} & \text{, otherwise} \end{cases} \quad (3)$$

$$a_{j_{score}} = \sum_{i}^{n} a_{j,i_{score}} \quad (4)$$

where, s_i represents, at the current iteration, the memory that kernel i needs so that it will run in hardware.

The first element is that, the gain of a kernel is only added to the score of a memory allocation when it can fit the available remaining SPM memory. This is represented by the if statement in Equation 3. In our example, and considering an SPM of size 3k, the gain of kernel $K3$ will not be added to the score of either memory block B or C, because its required memory does not fit the SPM.

The second change to the AMMA algorithm is represented by the equation $k_{i_{gain}} \cdot \frac{a_{j_{size}}}{s_i}$. This equation multiplies the kernel gain by the proportion the current block represents of the total memory needed for kernel k_i. This way we take into account how much the current memory block consumes of the total required memory the kernel needs. This is represented by the $\frac{a_{j_{size}}}{s_i}$. The larger this factor is, the more the kernel gain is taken into account.

The modified algorithm, that recomputes the scores after each allocation, is shown in Figure 8. Assuming f is the free memory at the current step, the new gain formula is given by Equation 4. The outline of the algorithm is given in Figure 8.

Assuming we have the kernels and memory allocations in Figure 5 and an available memory of 5 kbytes, the steps of AMMA and AMMAe are shown in Table I and Table II.

TABLE I: AMMA algorithm example

Iteration	Memory allocations scores				Allocated to SPM	Free SPM
	A	B	C	D		
1	200	400	510	310		5
2	200	400	-	310	C	4
3	200	-	-	310	C, B	3
4	-	-	-	310	C, B, A	1
Kernels executed accelerated: K1,K2 - gain 400ms						

For AMMA the memory allocation's scores are computed only once using Equation 1.

The gain for memory block A is the one generated by kernel $K1$, A while the gain for memory block D is the one generated by kernel $K1$. For memory blocks C and D, 2 kernels contribute to their gains. In each step, the memory block with the highest score is chosen to be allocated to the SPM. The

```
allocate_to_DDR(A,B,C,D)    allocate_to_SPM(A,B,C)
                            allocate_to_DDR(D)
........................    ........................
call K1(A,B)                call accelerated K1(A,B)
call K2(B,C)                call accelerated K2(B,C)
call K3(C,D)                call K3(C,D)

Gain:0                      Gain : 400 ms
```

Fig. 9: Code before and after AMMA algorithm

instructions that would be executed with and without AMMA applied are shown in Figure 9. We can see memory blocks A, B, C are allocated to SPM and $K1$ and $K2$ are accelerated.

TABLE II: AMMAe algorithm example

Iteration	Memory allocations scores gains				Allocated to SPM	Free SPM
	A	B	C	D		
1	133	166	177	232		5
2	133	166	410	-	D	2
3	0	200	-	-	D, C	1
4	0	-	-	-	D, C, B	1
Kernels executed accelerated: K2,K3 - gain 510ms						

```
allocate_to_DDR(A,B,C,D)    allocate_to_SPM(B,C,D)
                            allocate_to_DDR(A)
........................    ........................
call K1(A,B)                call K1(A,B)
call K2(B,C)                call accelerated K2(B,C)
call K3(C,D)                call accelerated K3(C,D)

Gain:0                      Gain : 510ms
```

Fig. 10: Code before and after AMMAe algorithm

AMMAe uses Equation 4 to compute the gain for each memory allocation. We give examples for s_{K1} and $a_{B_{score}}$ as the rest are computed similarly. The memory needed to run $K2$ accelerated is:

$$s_{K1} = a_{A_{size}} + a_{B_{size}} = 3$$

We compute now the score associated with block B. B is used by two kernels, $K1$ ($K1_{gain} = 200$) and $K2$ ($K2_{gain} = 200$). But kernel $K1$ needs both blocks A ($a_{A_{size}} = 2k$) and B ($a_{A_{size}} = 1k$) to run, so, we add to the score of B just a part of gain of kernel $K1$ proportional to block size of B from the total size needed ($s_{K1} = 3$). The same applies for $K2$. The entire equation is:

$$a_{B_{score}} = k_{K1_{gain}} \cdot \frac{a_{B_{size}}}{s_{K1}} + k_{K2_{gain}} \cdot \frac{a_{B_{size}}}{s_{K2}} = 166$$

After each allocation iteration, all the scores have to be recomputed as the total sizes needed by one kernel will change (we consider just the additional size needed, without taking into account the already allocated blocks). After the first iteration, the total size needed by kernel $K3$ will be just the size of C, as block D is already allocated. Hence the score for C will change. The algorithm is applied until no further allocation is possible.

The instructions that would be executed without and with AMMAe applied are shown in Figure 10. We can see B, C, D are allocated to SPM and $K2$ and $K3$ are executed accelerated, resulting in a bigger gain than for AMMA.

ILP formulation

For each of the memory allocation we associate a 0 - 1 variable (x_j) which will be 1 in case that the memory allocation will be made in a SPM. For each combination of memory allocations (so, for each kernel invocation) we associate a 0 - 1 variable (y_i) which will be 1 in case all the memory allocations in it are allocated in SPM. Let n be the number of memory allocations combinations. Using this notation our objective function is:

$$max(\sum_{i=1}^{n} y_i \cdot k_{i_{gain}}) \quad (5)$$

We need a constraint linking the combinations of memory allocations with each allocation. Let c_i be the number of allocations in C_i. The idea is that y_i is 1 just if all the corresponding x_j are also 1. We can e express this for each i, as:

$$y_i \le \frac{\sum_{a_j \in C_i} x_j}{c_i} \quad (6)$$

Let m be the total number of memory allocations. Ensuring that all the allocation fit in SPM will be imposed by the following constraint (where MEM is the total size of SPM):

$$\sum_{j}^{m} x_j \cdot a_{j_{size}} < MEM \quad (7)$$

VII. Empirical validation

Although the analysis was done for the HARTES platform, we did the empirical validation of our approach on different but similar platform namely a Xilinx Virtex-4 ML410 which is based on the Xilinx XC4VFX60 FPGA. The main reason for this was that the development of the operating system for HARTES was not finished at the time of the tests. The memories used in our design are: a Flash memory used as external memory, an internal 256 MB DDR2 memory as main memory and a 128 kbytes BRAM used as scratch pad memory. The kernels were implemented using the DWARV tool [13]. The system runs at 200 MHz and the hardware designs are clocked at 100 MHz. The compiler used to instrument the application was a modified GCC 4.3 Power-PC compiler.

As this is a runtime algorithm, **overheads** are important in measuring its performance. We present separately the overheads involved in constructing the MAL in Table III. The videos used are the videos available at [14]. Compared to the speedups that can be obtained by applying AMMA, the overhead incurred by it is negligible.

TABLE III: Execution time overhead of constructing memory allocation table for stack and dynamically allocated variables

Instrumen -tation	1	2	3	4	5	6
Dynamic	0.38%	0.37%	<0.1%	<0.1%	<0.1%	0.24%
Stack	0.36%	0.58%	0.33%	1.07%	<0.1%	0.52%

Another, similar, but less important issue is the execution time of the AMMA and AMMAe algorithms. Even if this algorithm will execute rarely, we present in Table IV the results obtained when testing with the synthetic applications. For the ILP algorithm, we ran just one subset of problems, because of the long execution times. The three cases are increasingly complex synthetic applications. Their generation is explained in Section VII-B.

A. H.264 video encoder

We used the instrumentation and algorithm presented to compile and run the H.264 video codec. The execution pattern in the encoder is dependent on the input data. Around 35% of the execution time is spent in two kernels: *satd_wxh* and

TABLE IV: Average algorithm execution times tested on the hardware platform (outside of the context of the applications)

Algorithm	Average execution times (ms)		
	Case 1	Case 2	Case 3
ILP	9980	30990	37780
AMMA	21	97	239
AMMAe	52	234	522

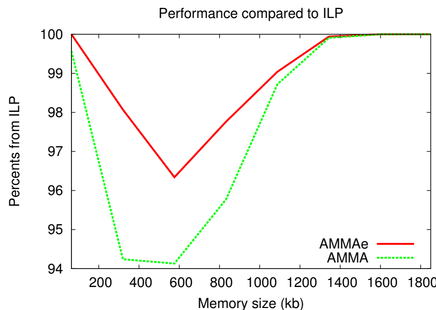

Fig. 11: Algorithm performance for different memory sizes

sad. The kernel hardware implementations offer a speedup of 2 and 1.8. The total memory used by both functions is 603 kbytes. The smallest block is 256 bytes, while the largest is 49 kbytes. The total available scratch pad memory is 128 kb. With this setup, by applying the AMMA algorithm we obtained a **performance improvement** of 14% compared to the GPP only execution. For this application and size of BRAM both AMMA and AMMAe gave the optimal solution. Assuming an infinite amount of BRAM the speedup that could be obtained is of 18%. The application contained 17 memory allocations that were used by the two kernels, and 29 memory allocation combinations.

B. Synthetic applications

Besides the test on the H.264 application we used benchmarks on synthetic applications, to test how far is our algorithm from the optimal solution. We considered in the synthetic applications that each memory allocation has a size between 128 bytes and 256 kilobytes. We use as reference the DWARV [13] hardware compiler which automatically generates the hardware kernels. The speedup obtained is up to 10X . We chose between 5 and 10 memory blocks and a large number of kernel (between 10 and 30) to test the algorithms in more complex situations than the one found in the motivational example. The results are seen in Figure 11.

The number of synthetic applications generated was 300. We compare against the ILP solution, which is optimal. The speedup obtained for each application depends on the amount of memory available and was in our tests between 2 and 6.

From the graph, we can see that in case there is little memory available, both algorithms perform as well as ILP. As the available memory increases, we can see that AMMAe is better than AMMA, within 4% of the ILP solution. Both algorithms converge to ILP when the available SPM-s are large enough to fit all the memory objects and all the kernels will be executed in hardware. These trends were also seen when varying the number of kernels and memory blocks (graphs omitted here for brevity), with AMMA being always within 14% of the ILP, and AMMAe being within 5% of the ILP solution.

VIII. Conclusions

In this paper, we presented the compiler driven AMMA algorithm that decides at runtime which is the best allocation for memory, taking into account the kernels that use the memory and as well as the gain that could be obtained by running those kernels on a dedicated hardware components. We showed how such a framework can simplify the development of applications while allowing a high flexibility by adapting to changing conditions. Our algorithm stays within 5% of the optimal solution given by an ILP formulation.

As future work, we will study the effect of parallel execution on the allocations and also the improvements that could be obtained by having various implementations for the same kernel.

References

[1] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte, "The molen polymorphic processor," *IEEE Trans. Comput.*, vol. 53, no. 11, pp. 1363–1375, 2004.

[2] O. Avissar, R. Barua, and D. Stewart, "An optimal memory allocation scheme for scratch-pad-based embedded systems," *Trans. on Embedded Computing Sys.*, vol. 1, no. 1, pp. 6–26, 2002.

[3] J. D. Hiser and J. W. Davidson, "Embarc: an efficient memory bank assignment algorithm for retargetable compilers," *SIGPLAN Not.*, vol. 39, no. 7, pp. 182–191, 2004.

[4] S. Udayakumaran and R. Barua, "Compiler-decided dynamic memory allocation for scratch-pad based embedded systems," in *CASES '03: Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*. New York, NY, USA: ACM, 2003, pp. 276–286.

[5] A. Bartzas, M. Peon-Quiros, S. Mamagkakis, F. Catthoor, D. Soudris, and J. M. Mendias, "Enabling run-time memory data transfer optimizations at the system level with automated extraction of embedded software metadata information," in *ASP-DAC '08: Proceedings of the 2008 conference on Asia and South Pacific design automation*. Los Alamitos, CA, USA: IEEE Computer Society Press, 2008, pp. 434–439.

[6] N. Nguyen, A. Dominguez, and R. Barua, "Memory allocation for embedded systems with a compile-time-unknown scratch-pad size," in *CASES '05: Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*. New York, NY, USA: ACM, 2005, pp. 115–125.

[7] A. Kannan, A. Shrivastava, A. Pabalkar, and J.-e. Lee, "A software solution for dynamic stack management on scratch pad memory," in *ASP-DAC '09: Proceedings of the 2009 Conference on Asia and South Pacific Design Automation*. Piscataway, NJ, USA: IEEE Press, 2009, pp. 612–617.

[8] A. Dominguez, S. Udayakumaran, and R. Barua, "Heap data allocation to scratch-pad memory in embedded systems," *J. Embedded Comput.*, vol. 1, no. 4, pp. 521–540, 2005.

[9] O. Ozturk, M. Kandemir, and I. Kolcu, "Shared scratch-pad memory space management," in *ISQED '06: Proceedings of the 7th International Symposium on Quality Electronic Design*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 576–584.

[10] S. Park, H.-w. Park, and S. Ha, "A novel technique to use scratch-pad memory for stack management," in *DATE '07: Proceedings of the conference on Design, automation and test in Europe*. San Jose, CA, USA: EDA Consortium, 2007, pp. 1478–1483.

[11] H. Cho, B. Egger, J. Lee, and H. Shin, "Dynamic data scratch-pad memory management for a memory subsystem with an mmu," in *LCTES '07: Proceedings of the 2007 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*. New York, NY, USA: ACM, 2007, pp. 195–206.

[12] [Online]. Available: http://www.hartes.org

[13] Y. D. Yankova, G. Kuzmanov, K. Bertels, G. N. Gaydadjiev, J. Lu, and S. Vassiliadis, "Dwarv: Delftworkbench automated reconfigurable vhdl generator," in *In Proceedings of the 17th International Conference on Field Programmable Logic and Applications (FPL07)*, August 2007, pp. 697–701.

[14] "Xiph.org test media," 2009. [Online]. Available: http://media.xiph.org/video/derf/