Configuration Management in Multi-Context Reconfigurable Systems for Simultaneous Performance and Power Optimizations*

Rafael Maestre, Milagros Fernandez
Departamento de Arquitectura de Computadores y Automática
Universidad Complutense de Madrid, SPAIN
http://www.dacya.ucm.es/maestre

Fadi J. Kurdahi, Nader Bagherzadeh, Hartej Singh
Department of Electrical and Computer Engineering
University of California, Irvine, USA

Abstract

In this paper, we present a novel solution to the problem of configuration management for multi-context reconfigurable systems targeting DSP applications, its goal being to minimize both, configuration latency and power consumption. We assume that this technique is applied within a larger compilation framework, which provides a scheduled task sequence of the considered application. Reconfiguration latency reduction is the first criteria to consider, and we prove that the optimal solution can be obtained in all cases. Secondly, power is optimized without affecting performance. The assumptions of the method are supported by the analysis of a mathematical model, and its effectiveness is demonstrated by some experiments.

1. Introduction

Reconfigurable computing combines a reconfigurable hardware processing unit with a programmable processor. The processor orchestrates the whole system and it may be also used for computations so as to contribute to the overall throughput. The reconfigurable hardware implements the core of computations. This kind of computing is consolidating as a viable design alternative to implement a wide range of computationally intensive applications. Its main advantage is clearly its versatility. These systems can be adapted to almost any target application, achieving an acceptable degree of performance in many cases. Most reconfigurable systems exploit its whole processing potential when executing applications with a lot of inherent parallelism. DSP and multimedia applications are some of those applications. This is why our work is focused on them. Additionally, this kind of applications have an internal structure with some features, such as regularity, that may be exploited in order to simplify the problem formulation.

The execution of such complex applications requires implementing consecutively different configurations during computation, which is called *run-time reconfiguration*. The term *dynamic reconfiguration* restricts this concept for the cases such that the reconfiguration overhead does not imply an essential computation stall. As the technology has produced devices with shorter reconfiguration times, the idea of dynamic reconfiguration has become a viable possibility. We could say that dynamic reconfiguration permits the consideration of a new dimension in the design space, time multiplexing of hardware resources. Instead of increasing the number of available resources, the configurations are swapped in and out of the actual hardware, as they are needed. Dynamic reconfiguration may be implemented in two different configuration memory styles. One of them allows selective access to the reconfiguration memory in a particular region of the reconfigurable device. At the same time the rest of the device remains fully operational. Xilinx XC6200 family is one example of this choice. In this kind of devices the portion of the hardware that is being reconfigured cannot be used. Recently, multi-context architectures have appeared to facilitate and accelerate the configuration switch [1]. They can store a set of different contexts (configurations) in a context memory. When a new configuration is needed, it is loaded from the context memory if available. As the context memory is on-chip, this operation is much faster than the reconfiguration from an external memory. For example, MorphoSys architecture [1] has a context memory that can store 32 different contexts, and a full change of context only takes one clock cycle, compared to one single context transfer from the external memory which takes more than 200 cycles. However, the context memory cannot store all the configurations of a specific application. Therefore, it is obvious that reconfiguration time may be dramatically reduced, but only if the context memory is carefully

* This work has been granted by Spanish Government Grant CICYT TIC 99-0474.

managed. We address the problem for multi-context architectures with an undefined number of contexts.

Besides performance, the minimization of power consumption is another very important issue that should be considered, at least for portable systems. Power in CMOS integrated circuits is mainly caused by the switching activity of internal capacitances. This means that power consumption can be reduced if we find a way to decrease the switching activity as much as possible in the whole chip. Although this kind of optimization can be performed at different compilation steps, we address this problem from the configuration scheduling and allocation point of view; therefore optimizations are physically restricted to the context (configuration) memory.

As typical applications are usually so demanding, performance is the first optimization factor to consider so that timing constraints are met. Additionally, the optimal solution for configuration minimizes the number of configuration swaps, which simultaneously produces a positive effect on power consumption. Then, the solutions with the same latency are explored to achieve further power improvements.

This work assumes that the different tasks that compose the target application have already been scheduled. This task scheduling can be obtained through the approach to scheduling in reconfigurable computing presented in [2]. It is the first step of a whole design development framework [3]. This scheduling technique provides the best task execution order, although the effect of the following tasks is only estimated. The current work is a part of the second step of this framework, and its goal is to accomplish the degree of performance that the scheduling task assumed, while optimizing power consumption.

Reconfigurable computing is an emerging area of computing that has opened many different research fields. For example, scheduling is an area that, although has been widely investigated in other fields, needs to incorporate the characteristics of this kind of systems. As a matter of fact, most approaches are versions of existing high-level synthesis techniques, extended to consider specific features of reconfigurable systems, such as the reconfiguration time [4]-[7]. More especially, context scheduling is a relatively new problem that has received little attention. To the best of our knowledge, [3] and [8] are the only work that explicitly tackles this issue. [8] addresses reconfiguration overhead reduction through configuration prefetch. However, its applicability is restricted to single context architectures. In [3] a heuristic solution to the current problem for multi-context architectures is presented. Other scheduling methodologies in reconfigurable computing, like [10,11], do not explicitly address this issue. Finally, we will mention that we have not found any previous work about power optimizations for multi-context systems.

The paper is organized into 6 sections. Section 2 introduces the relevant issues, and provides a first approach to the problem. Section 3 analyzes the minimization of context loading, which improves both performance and power consumption. Section 4 is devoted to the allocation of memory replacements, whose goal it to achieve the lowest power consumption for the optimal performance scheduling. Then, section 5 summarizes the experimental results that have been obtained. Finally, section 6 concludes the paper.

2. Problem Overview

A typical DSP or multimedia application is composed of a sequence of macro-tasks that are repeatedly executed as a loop. We use the term kernel to refer to one of those well-defined macro-tasks [2]. Some examples of those applications are MPEG (video compression and decompression), JPEG (image compression) and ATR (Automatic Target Recognition), whereas DCT (Discrete Cosine Transform) or ME (Motion Estimation) may exemplify some of these kernels. Hence, in this paper we will assume that any kernel sequence is periodic. Moreover, as the kernels of those applications can be scheduled through the algorithm presented in [2], the resulting kernel scheduling constitutes the input to our problem.

In order to provide an intuitive approach to the problem, we will consider the examples shown in Fig. 1. The figure represents the status of the CM (Context Memory) by means of some CM snapshots taken just before the beginning of a kernel execution. We have represented one single inner iteration of the periodic sequence. A whole kernel execution implies the implementation of a specific sequence of contexts, which will be assumed to be in the CM during its corresponding kernel execution. As a kernel always uses the same contexts, it may be possible to keep some of them in the memory, and reuse them in the next kernel execution. We will use the term *static* to refer to those contexts. On the other hand, the rest of the context will have to be loaded before the beginning of the next execution, and they will be stored in the remaining part of the CM. These contexts will be called *dynamic contexts*. Thus, dynamic contexts imply loading time, whereas static contexts save loading time, and hence we should maximize the static part of the CM. For example Fig. 1 illustrates two solutions with different static sizes, and L_T (total number of context loadings) is bigger for the example with less static contexts (Fig. 1.a).

In subsequent sections we will prove that the static part is maximized, and thus L_T is minimized, when the numbers of loadings for all kernels (all L_i) are as equal as possible, but first we will provide a brief idea here.

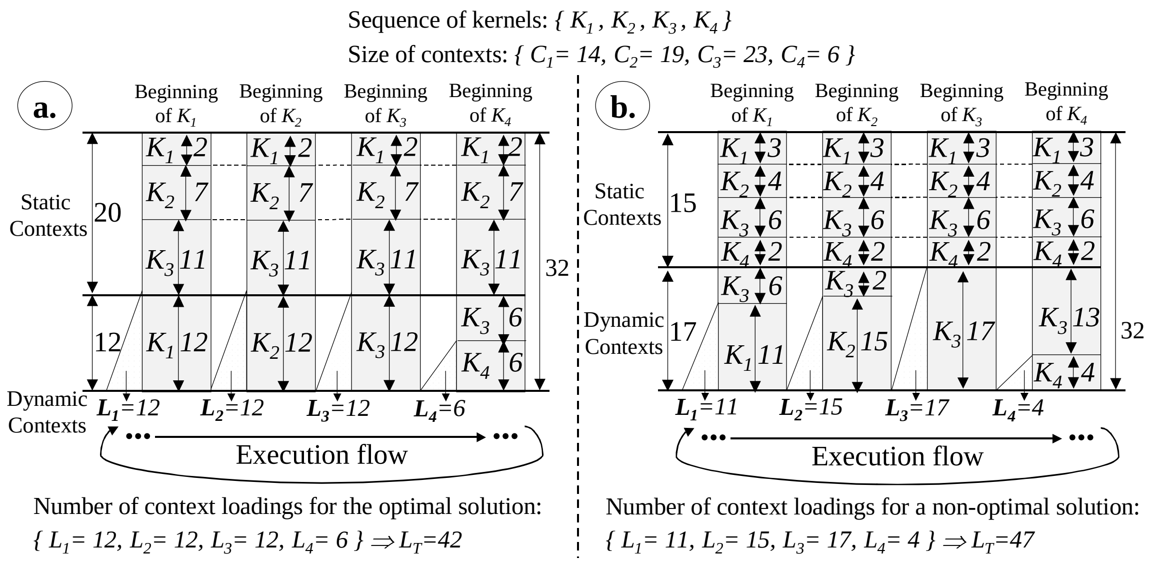
Sequence of kernels: { K1 , K2 , K3 , K4 }
Size of contexts: { C1= 14, C2= 19, C3= 23, C4= 6 }
a.
b.
Beginning of K1
Beginning of K2
Beginning of K3
Beginning of K4
Static Contexts
Dynamic Contexts
20
12
32
15
17
L1=12 L2=12 L3=12 L4=6
L1=11 L2=15 L3=17 L4=4
Execution flow
Number of context loadings for the optimal solution:
{ L1= 12, L2= 12, L3= 12, L4= 6 } ⇒ LT=42
Number of context loadings for a non-optimal solution:
{ L1= 11, L2= 15, L3= 17, L4= 4 } ⇒ LT=47

Fig. 1. CM snapshots for two possible context loading schedules. The snapshots have been taken just at the beginning of each kernel execution, and show the content of the CM positions. K_i stands for the kernel "*i*". L_i represents the number of loadings regarding kernel K_i, and L_T is the total number of context loadings. C_i is the number of contexts for kernel K_i. The triangles depict the loaded contexts.

Imagine one of those solutions, like the one presented in Fig. 1.a. If L_i were bigger for at least one kernel, this would reduce the size of the static part, and therefore some static contexts would become dynamic and should be later reloaded. This fact would certainly increase the total number of loadings. Fig. 1.b represents a solution such that L_i substantially different for every kernel. As shown the solution is worse. In this example the criteria to build this solution has been that the number of loadings for a kernel is proportional to its context size.

Similarly, if context loadings are accomplished too soon, they will occupy CM positions, which will reduce the number of static positions. Therefore, context loadings have to be performed *as late as possible* (*ALAP*).

The minimization of context loadings optimizes reconfiguration latency, but at the same time it potentially improves the power consumption, since the total number of transitions from "0" to "1" of the internal capacitance associated to individual CM cells is kept within low values. Once the solution with the lowest latency has been generated, power can be further optimized. So far we have not considered context allocation, which will directly influence power. The number of bit changes that implies the consecutive loading of two configurations in the same CM position determines the actual power consumption, and there are many different configuration replacements for a given schedule. For example, in Fig. 1.a K_1 contexts can replace any of K_3 or K_4 contexts. The exploration of the alternatives will lead to different power numbers.

All the previous ideas are analyzed in detail in the following sections.

3. Analysis of Context Loading Minimization

In this section we present the mathematical analysis that leads us to the optimal solution in terms of number of context loadings, which optimizes performance, besides reducing power consumption. The reasoning is based on the fact that the size of the CM (*SCM*) is a physical constraint that has to be fulfilled in all cases. It is obvious that the number of both, dynamic and static contexts cannot exceed *SCM*. We suggest the reader to use Fig. 1 in order to visualize the meaning of the expressions.

The dynamic part of the CM (*Dyn*) is given by the maximum number of context loadings (L_i):

$$Dyn = \underset{\forall i}{MAX}\left(L_i\right)\cdot \qquad (1)$$

On the other hand, the static part of the CM (*Stc*) is composed of the contexts that are not loaded every iteration (C_i - L_i), thus if C_i is the number of contexts for kernel K_i,

$$Stc = \sum_{\forall i}\left(C_i - L_i\right)\cdot \qquad (2)$$

Therefore, the total size of the context memory is:

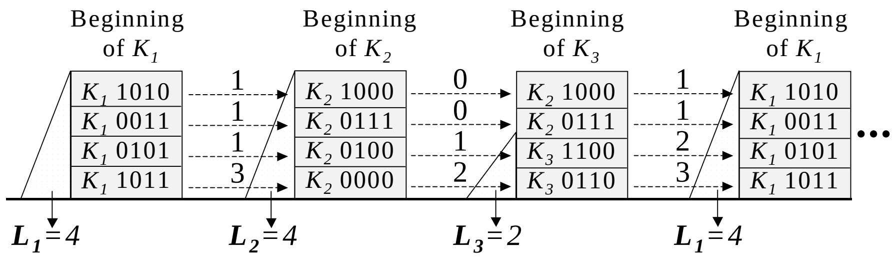

Fig. 2. Snapshots of the CM dynamic part for a possible scheduling showing some hypothetical allocation of contexts of 4 bits. The dotted arrow indicates the number of bit changes for a context allocation.

$$SCM \geq Dyn + Stc = \underset{\forall i}{MAX}(L_i) + \sum_{\forall i}(C_i - L_i) \cdot \qquad (3)$$

From (**3**) we can express the total number of loadings as:

$$L_T = \sum_{\forall i} L_i \geq \underset{\forall i}{MAX}(L_i) + \sum_{\forall i} C_i - SCM \cdot \qquad (4)$$

This expression proves the idea introduced in the previous section. If L_i is bigger for a particular kernel, K_i, than L_j for the others, this one will determine the lowest feasible value of L_T. Hence, if L_i for all kernels take the same value and expression (**4**) is fulfilled, this will be the optimal solution in terms of reconfiguration latency. It should be noted that as the number of context loadings for a kernel cannot exceed its number of contexts (that is $C_i \geq L_i$), all L_i have to be at least as equal as possible. For example in Fig. 1.a as $C_4 \geq L_4$ the maximum value for L_i is 4. However, the rest of the kernels can fulfill the requirements, and thus it minimizes the total number of loadings, L_T. Thus, for the example in Fig. 1.a:

$$\begin{aligned} L_T &= 42 \geq \underset{\forall i}{MAX}(L_i) + \sum_{\forall i} C_i - SCM = \\ &= MAX(12,12,12,6) + 62 - 32 \end{aligned} \qquad (5)$$

If any L_i is increased L_T will also be increased, and hence the example in Fig. 1.a is the optimal solution.

In order to obtain the best L_i distribution, we will make the assumption that all the kernels may have equal values of L_i, $A = L_i$, which means that $MAX(L_i) = A$, and $L_T = N_T{*}L_i$, where N_T is the total number of kernels. Consequently, from (**4**):

$$L_i = \frac{\sum_{\forall i} C_i - SCM}{N_T - 1} \cdot \qquad (6)$$

If some kernel/s cannot satisfy this expression, we will choose the lowest C_i and its L_i will take the highest possible value, as in the example of Fig. 1.a. Then, this kernel will not be considered in the next computation of equation (**6**). This process is repeated until a feasible solution is obtained. As the reader may imagine, the complexity of this algorithm is very low, and it can easily handle sets with a big number of kernels, generating the optimal solution in very short times.

4. Allocation for Power Optimization

The solution presented in section 3 minimizes the total number of loadings, and consequently not only optimizes reconfiguration latency, but also reduces the number of context replacements, which improves power consumption. At the level of abstraction we are working on, power can only be reduced by minimizing the number of bit level transitions of the CM cells. Thus, we have to explore context allocation alternatives, in order to find the configuration replacements with the lowest number of bit changes. Fig. 2 illustrates the allocation of a bit specification of dynamic contexts for a small example of three kernels and a CM with contexts of 4 bits.

The only way to evaluate the power consumption that a specific replacement implies is to count the number of bit changes between contexts. At first sight, it may seem that the number of combinations is too high, since we should obtain the bit changes between all contexts. In a solution of the type proposed in section 3, most kernels have the same number of context loadings, and these contexts fill the whole dynamic part of the CM. Therefore, it is very common that a context can only replace the contexts that belong to the previously executed kernel, which reduces the number of potential combinations. For example in Fig. 2, K_2 contexts can only replace K_1 contexts. However, for some kernels there may be some additional possibilities. Thus, in Fig. 2 the number of K_3 context loadings is lower than the size of the dynamic part. This implies that the contexts of the following kernel, K_1, can replace either K_2 or K_3 contexts, what introduces additional choices. To generalize this idea, we could say that a given context can replace any previously executed one, only if the dynamic part of the CM is not completely used between the corresponding kernel executions, like between kernels K_2 and K_1. In this case we will say that there is a *direct path*

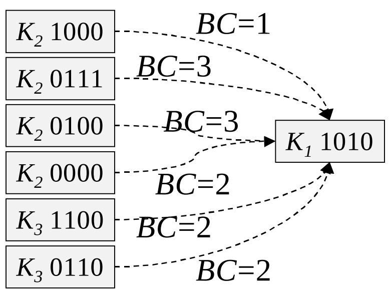

Fig. 3. Evaluation of CM possible replacements of Fig. 2, between one K_1 context and the contexts with a *direct path*. *BC* is the number of bit changes between contexts.

between contexts (which can also be applied to consecutive kernels).

The procedure that we propose requires computing *BC* (number of bit changes) between all the contexts with a *direct path*. Fig. 3 shows the possible direct paths and its *BCs* to one K_1 context. *BC* characterizes each replacement, and can be used as a guideline to build the final allocation. In this way, if the replacements with the lowest *BC* are consecutively allocated we will obtain a solution with a low total *BC*. This *BC* will be optimal if it is composed of all the replacements with lowest *BC*. However, we might think that sometimes the optimal set of replacements cannot be obtained in this manner, because the selection of some particular replacements might prevent some future selections that may produce an overall *BC* reduction. In order to check this kind of solutions we could explore some replacement allocations that produce a *BC* increase. Then, if no improvement is obtained later, these replacements will be discarded.

Fig. 4 illustrates the exploration process. First, the replacements are numbered in ascending order in accordance to *BC*. The replacements with the lowest number (that can be allocated) are evaluated up to a certain depth. For example in Fig. 4 the first branch of the exploration tree is explored to a depth of 3, and it is composed of the replacements {*1,2,5*}. Note that it is assumed that the allocation of replacements *1* and *2* prevent the allocation of *3* and *4*, which forces to allocate *5*. Then, we check for some solutions that might improve the result. Thus, in Fig. 4 we now consider the branch {*1,3,4*}, which in some cases may improve the result of the first branch. It depends on the actual values of *BC*. It should be noted that it is not necessary to explore any branch that begins with {*2,...*}, since the best possible branch would be {*2,3,4*}, which is certainly equal or worse that any of the two explored. All the elements of {*2,3,4*} are equal or worse than {*1,3,4*}. This consideration dramatically reduces the exploration time, since the exploration of all the branches may imply a

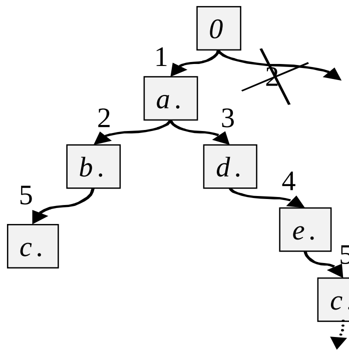

Fig. 4. Exploration tree example. The exploration *depth* is assumed to be *3*. The replacements have been numbered in ascending order of BC, and the edge labels stand for the explored replacements.

combinatorial explosion even if carried out up to a small depth. After that, the best replacement (*1* in the figure) is allocated and the exploration continues in the same way.

As the exploration depth grows the probability of finding a better solution increases. In the limit, when the depth equals the number of possible replacements, the optimal solution is always found. However, the experience shows that the exploration time grows exponentially as the depth increases, which makes unfeasible to perform the optimal exploration in medium and big size problems. The higher depth that can be handled is typically around *15*.

It should be noted that if the tree is visited from the top to any leaf, it is not possible to find a descending edge sequence. This fact guarantees that every solution is only generated once. For example, the replacement sequence, *{4,1,3}*, will never be generated, as it is equivalent to the sequence *{1,3,4}*.

5. Experimental Results

We have carried out a series of experiments in order to demonstrate the effectiveness of the proposed techniques. We have chosen MorphoSys [11-12] as the target multi-context system in order to use real system parameters. MorphoSys has a configuration memory of 32 contexts, and each one is composed of 8 words of 32 bits. Therefore, each context has a total of 256 bits. This data has been used in order to generate the experiment information as explained below.

The experiments are formed of three real applications, as well as a set of randomly generated ones. MPEG is a standard for video compression, while ATR stands for Automatic Target Recognition and we consider its two main tasks SLD (Second Level Detection) and FI (Final Identification). Both tasks, SLD and FI, have the same number of kernels and contexts, and therefore we present

Experimental data	Proposed solution (All optimizations)				Only L_T optimi-zations	Only allocation optimizations		Without optimi-zations
	L_T	*BC*	Exploration time		*BC* (*RI*)	L_{Ti} (*RI*)	*BC* (*RI*)	*BC* (*RI*)
			Depth=1	Depth=12				
Ex1: *N_T=3; C_i={10,15,25}*	27	2448	0.01s	19.78s	3160 (29%)	40 (48%)	3504 (43%)	4756 (94%)
Ex2: *N_T=4; C_i={26,15,30,17}*	80	7208	0.12s	18.71s	9372 (30%)	84 (5%)	7578 (5%)	9850 (37%)
ATR (SLD** and **FI): *N_T=4; C_i={24,24,24,12}*	72	6882	0.06s	2.37s	8724 (27%)	84 (17%)	8104 (18%)	10178 (48%)
Ex3: *N_T=5; C_i={20,5,7,18,3}*	28	2998	0.02s	19.36s	3472 (16%)	51 (82%)	5226 (74%)	6330 (111%)
Ex4: *N_T=6; C_i={8,10,16,3,4,21}*	38	3978	0.02s	9.69s	4771 (20%)	57 (50%)	5890 (48%)	7228 (82%)
Ex5: *N_T=7; C_i={25,8,10,2,9,11,6}*	47	5192	0.02s	7.68s	5767 (11%)	57 (21%)	6238 (20%)	6980 (34%)
MPEG: *N_T=7; C_i={8,4,21,6,6,21,4}*	48	5108	0.05s	41.48s	5928 (16%)	70 (46%)	7272 (42%)	8572 (68%)
Ex5: *N_T=8; C_i={10,5,9,3,8,12,20,2}*	44	4954	0.02s	15.34s	5582 (13%)	61 (39%)	6808 (37%)	7784 (57%)
Ex6: *N_T=12;C_i={10,5,12,2,15,1,7,9,22,1,3,8}*	78	9575	0.18s	430.74s	11084 (16%)	92 (18%)	10641 (11%)	12844 (34%)
Ex7: *N_T=15 C_i={15,2,8,19,7,9,17,1,25,13,8,2,1,6,8}*	129	15347	0.26s	161.04s	17704 (15%)	139 (8%)	16038 (5%)	19010 (24%)
Ex8: *N_T=20 C_i={8,7,12,20,4,2,17,5,25,7,4,24,3,6,8, 4,6,5,10,20}*	184	20794	0.7s	92.31s	23414 (13%)	196 (7%)	21971 (6%)	24926 (20%)

L_T = Total number of context loadings. *BC*= Number of bit changes. *RI*= Relative increment of BC with respect to the proposed solution when some of the optimizations are not performed.

Table 1. Experimental results.

the results for only one experiment. As shown in Table 1 the proposed methodology generates similar results for both, real and synthetic experiments.

The numbers of contexts in the real applications (for example {*8,4,21,6,6,21,4*} for MPEG) correspond to real data, however all the configuration patterns have been randomly generated, since we do not know its actual form. In a first approach all the bits were independently generated. This led us to a completely random structure such that any replacement implied almost the same number of bit changes. As this is not the real case, we decided to change the configuration generation procedure. In a system like MorphoSys each 32-bit context word configures the functionality of one single reconfigurable cell. The structure of a context word is not completely random, and keeps within certain ranges. Moreover, a particular functionality of a cell may be later reused by a later implemented context. The replacement of these two cell configurations would produce no power consumption. Consequently, first we randomly generated a big number of 32-bit context words within the architectural constraints, and then configurations were randomly composed of the generated context words. This enables us to generate experiments with similar features to real applications.

In order to demonstrate that the proposed scheduling technique always generates the optimal solution for context scheduling, we implemented an exhaustive search algorithm, which can explore the whole search space for medium and small size examples (lower than 7). The results for both methods match in all those cases, however the exploration time is order of magnitudes smaller for our algorithm.

Table 1 and Fig. 5 summarizes the experimental results we have obtained. In order to compare the configuration latency and power consumption improvements, we present the number of context loadings (L_T) and bit changes (*BC*) for the proposed technique, as well as for three greedy approaches that do not perform some of the proposed optimizations.

1. The first algorithm only performs *context loading* (L_T) *optimizations* (column 6 in Table 1 and white bars in Fig. 5). *BC* is obtained when allocation is not considered. The relative increments are within 12-30% worse than the proposed approach.
2. The results presented in columns 7-8 and the gray bars in Fig. 5, show for only *allocation optimizations*. In this case we assume that contexts are loaded when necessary. Now *BC* increments range from 5% to 74%. Similarly, L_T increments range form 5 to 82%.

3. Finally, we present the results for the first generated solution when no optimization is considered (column 9 in Table 1 and black bars in Fig. 5). *BC* increments are now really big: 20-111%. L_T is the same as in point 2.

In all cases the performance and power improvements are clearly significant.

We have carried out all experiments for a depth range from 1 to 12, and the impact on the final result is negligible. The reason is that most contexts replace just executed contexts, which does not prevent the allocation of other replacements. The replacement with lowest *BC* can almost always be allocated leading to a quasi-optimal solution. In the table we show the exploration time for two different values of depth. The algorithm is carried out in a really short exploration time for a *depth= 1*. We could say that the proposed technique provides substantial performance and power improvements in very short exploration times.

6. Conclusions and Future Work

In this paper we have presented a new approach to the problem of context scheduling and allocation for multi-context reconfigurable systems targeting DSP and multi-media applications. The goals of this approach are to minimize configuration latency and power consumption. The analysis of a mathematical model enabled us to deduce an algorithm for context scheduling that generates the optimal solution in all cases. We also propose a technique to efficiently explore context allocation possibilities, which leads to an overall power improvement without affecting the achieved system performance. The effectiveness of the proposed work is analyzed in the experimental results section. The results for real application and synthetic experiments validate our assumptions, and show the quality of the technique. Future work will explore the power improvements that can be obtained when we additionally consider configuration latency increases.

References

[1] G. Lu, H. Singh, M. Lee, N. Bagherzadeh, F. J. Kurdahi, E. M. C. Filho, “MorphoSys: An Integrated Re-configurable Architecture”, Proceedings of the 5th International Euro-Par Conference Toulouse, France, August/September 1999, pp. 727-734

[2] R. Maestre, F. J. Kurdahi, N. Bagherzadeh, H. Singh, R. Hermida, M. Fernandez, “Kernel Scheduling in Reconfigurable Computing”, DATE Proceedings, pp. 90-96, 1999.

[3] R. Maestre, M. Fernandez, R. Hermida, N. Bagherzadeh, “A Framework for Scheduling and Context Allocation in Reconfigurable Computing”, International Symposium on System Synthesis Proceedings, San Jose, California, USA, pp.134-140, November 1999.

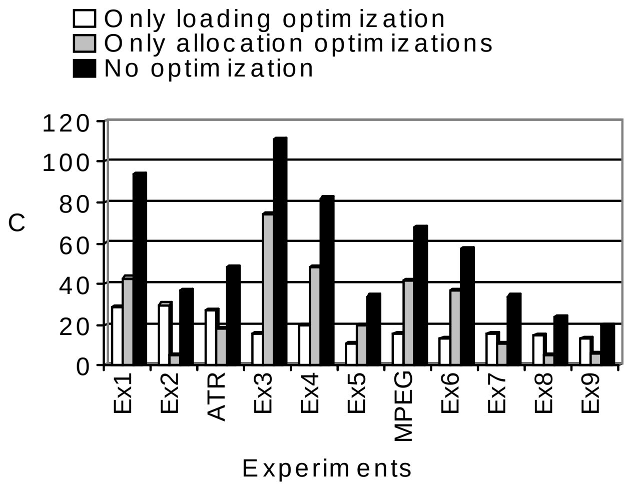

Fig. 5. Relative increment of BC when the proposed optimizations are not performed.

[4] I.Ouaiss, S. Govindarajan, V. Srinivasan, M. Kaul and R. Vemuri, "An Integrated Partitioning and Synthesis system for Dynamically Reconfigurable Multi-FPGA Architectures", 5th Reconfigurable Architectures Workshop, 1998 (RAW'98).

[5] M. Vasilko and D. Ait-Boudaoud, "Architectural Synthesis Techniques for Dynamically Reconfigurable Logic", 6th International Workshop on Field-Programmable Logic and Applications, FPL '96 Proceedings, pp.290-296.

[6] M. Vasilko and D. Ait-Boudaoud, "Scheduling for Dynamically Reconfigurable FPGAs", in Proceeding of International Workshop on Logic and Architecture Synthesis, IFIP TC10 WG10.5, Grenoble, France, Dec. 18-19, 1995, pp. 328-336.

[7] K. M. GajjalaPurna, D. Bhatia, "Temporal partitioning and scheduling for reconfigurable computing", Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines, 1998, pp. .329-330.

[8] S. Hauck, “Configuration Prefetch for Single Context Reconfigurable Coprocessors” ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, pp. 65-74, 1998.

[9] K. M. GajjalaPurna, D. Bhatia, "Temporal partitioning and scheduling for reconfigurable computing", Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines, 1998, pp. .329-330.

[10] M. Kaul and R. Vemuri, “Temporal Partitioning Combined with Design Space Exploration for Latency Minimization of Run-Time Reconfigured Designs”, Design, Automation and Test In Europe Proceedings, 1999 (DATE’99), pp. 202-209.

[11] H. Singh, M. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, T. Lang, R. Heaton and E. M. C. Filho, “MorphoSys: An Integrated Re-configurable Architecture”, Proceedings of the NATO Symposium on System Concepts and Integration, Monterey, CA, April 1998.

[12] H. Singh, N. Bagherzadeh, F. J. Kurdahi, G. Lu, M. Lee, E. Chaves, R. Maestre, “MorphoSys: Case Study of a Reconfigurable Computing System Targeting Multimedia Applications”, Design Automation Conference Proceedings, DAC-2000.