Auto-SI: An Adaptive Reconfigurable Processor with Run-time Loop Detection and Acceleration

Tanja Harbaum*, Christoph Schade*, Marvin Damschen†, Carsten Tradowsky‡, Lars Bauer†, Jörg Henkel† and Jürgen Becker*
*Institute for Information Processing Technology (ITIV), Karlsruhe Institute of Technology (KIT)
Email: {harbaum, becker}@kit.edu, christoph.schade@student.kit.edu
†Chair for Embedded Systems (CES), Karlsruhe Institute of Technology (KIT)
Email: {damschen, lars.bauer, henkel}@kit.edu
‡FZI Forschungszentrum Informatik
Email: Carsten.Tradowsky@fzi.de

***Abstract*—Modern computer architectures have an ever-increasing demand for performance, but are constrained in power dissipation and chip area. To tackle these demands, architectures with application-specific accelerators have gained traction in research and industry. While this is a very promising direction, hard-wired accelerators fall short when too many applications need to be supported or flexibility is required. In this paper, we propose an automatic loop detection and hardware acceleration approach for an adaptive reconfigurable processor. Our contribution is Auto-SI, an automated process that transparently and dynamically provides hardware acceleration alongside a general-purpose processor by employing reconfigurable hardware. We detail the benefits of Auto-SI, i.e., transparent and flexible acceleration of unmodified binaries, provide an analysis of the overheads incurred and present an evaluation of our implementation prototype.**

I. Introduction

One of the major industry trends in the field of processor architectures is adding application-specific circuits (accelerators) for all sorts of applications on a single chip. The main reason for this development is to increase the energy efficiency of the system, e.g., increasing the performance per watt for accelerated applications. These accelerators can provide good performance per watt, but are costly in terms of chip area. Consequently, they present a pragmatic solution that is limited to markets in which chip area is not a higher priority constraint such as desktop computers or servers.

In mobile and embedded systems, the chip's physical size is often one of the main constraints. When adding numerous specialized accelerators for several applications, the area allocated to accelerators often exceeds the area allocated to the processors that are employed in this domain. Therefore, adding application-specific accelerators is only reasonable up to a certain extent [1]. So, in contrast to desktop computers and servers, the metric of performance per area is one of the higher priority optimization goals in embedded systems. Application-specific accelerators often have a very low utilization at run time, providing very low performance per area for the system. Therefore, the desire to increase performance per area is contradictory to the increasingly widespread use of specialized accelerators. A logical consequence would be to use more general accelerators so that they can be used more often, resulting in increased performance per area.

While less specific accelerators provide a lot of desired benefits, they perform worse than more specialized accelerators when considering performance only (more specialization generally means more operations that the accelerator can cover and optimize for, therefore increased performance). To achieve both advantages –strong specialization and general applicability– our concept of dynamic acceleration is introduced. The aim of dynamic acceleration is to adapt an accelerator for a very specific computation, while keeping the range of applications to which it can adapt to as wide as possible. To accomplish this, we observe instructions executed by the CPU at run time to find compute-intensive kernels and extract the corresponding data flow graph (DFG). From the DFG, an accelerator bitstream is created (assembled out of smaller bitstream chunks that implement very general operations) and configured onto reconfigurable hardware (FPGA). Finally, once the reconfiguration has completed, the kernel's most compute-intensive instructions are replaced with a call to the accelerator. In essence, this results in performance gains so far only achieved with a specialized accelerator and the applicability of a general accelerator.

Our concept provides implications beyond the embedded systems domain. Virtually all processors available on the market today provide accelerators of some sort that are interfaced using instruction set extensions (ISEs). ISEs require specific knowledge by the software developer or the compiler about their availability and functionality. In consequence to achieve hardware acceleration, steps ranging from recompilation to complete software redesign may be required. Unless software is adapted for the specific accelerators that are available on a target processor, the accelerators are not utilized but consume area and power. To relieve developers from the task of employing specialized hardware accelerators and support legacy software, transparent acceleration is a key factor.

Our contribution in this work is *Auto-SI*: an automated process that transparently and dynamically provides hardware acceleration alongside a general purpose processor (GPP) using reconfigurable hardware. As such, any program written in the GPPs instruction set can be run and potentially benefit from Auto-SI. Our concept allows past, present or future software to

978-1-5386-4034-0/17/$31.00 ©2017 IEEE

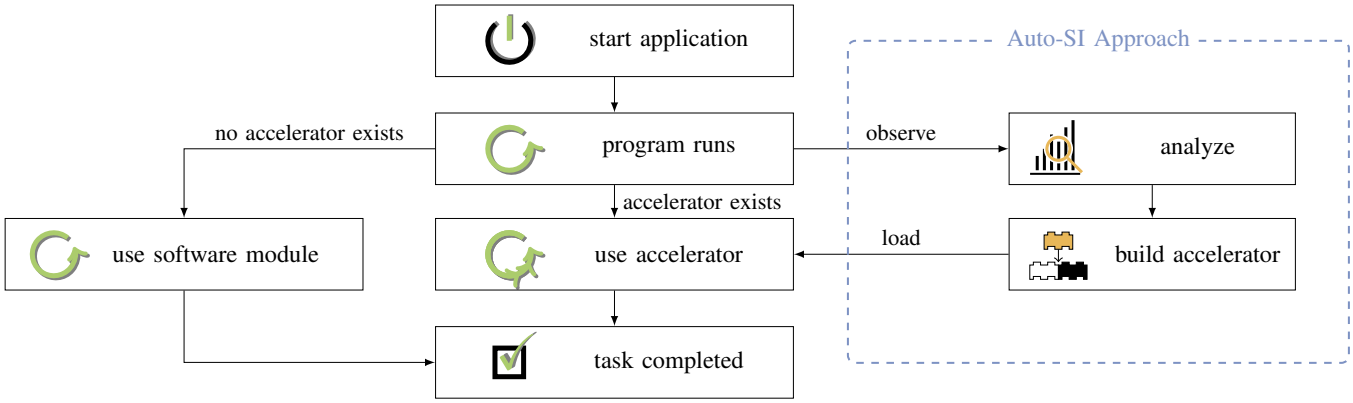

Figure 1: Idea of the Auto-SI Approach

make use of hardware acceleration.

This paper is structured as follows: In Section II, other approaches to dynamic and/or transparent hardware acceleration are discussed, leading to the presentation of our concept in Section III. To demonstrate the potential gains, an implementation was designed; it is described in Sections IV and V. Finally a conclusion with ideas for further work is provided in Section VI.

II. State of the Art

In the field of reconfigurable hardware acceleration, two main concepts have influenced research activity.

In the first, researchers are using an FPGA as underlying hardware and an offline-profiler to generate required accelerators for it [2, 3]. This approach is not transparent and as such requires work on existing binaries to utilize proposed acceleration concepts. The second main concept as presented in [4–6] provides total transparency. This is achieved by using coarse-grained reconfigurable hardware (so-called coarse-grained reconfigurable arrays, CGRAs). Generation of configurations as well as reconfiguration takes considerably less time in this approach compared to an FPGA-based solution, because CGRAs consist of functional units that provide a fixed set of operations on fixed word widths (that generally map naturally to operations performed by the CPU). FPGAs provide increased flexibility and opportunities for optimization (arbitrary logical functions on custom word widths) at the cost of additional reconfiguration delay and a more challenging process to generate configurations.

Lysecky et al. proposed the Warp acceleration approach that would meet the requirements of transparent and dynamic acceleration [7]. In their approach, they have added a profiler module, an on-chip CAD module as well as an FPGA to an existing processor design. The profiler monitors the instruction stream processed by the processor with regard to critical kernels. Upon detection, these are passed to an on-chip CAD module, which performs partitioning, synthesis, mapping, placing and routing in order to re-implement the software kernel as a hardware module on the FPGA. Once the hardware module has been loaded onto the FPGA, the binary is changed so as to call the accelerator instead of running the software kernel. In fact, the "on-chip CAD module" is a dedicated processor, that still requires several seconds (or longer) to create an equivalent accelerator configuration. In relation to the main processor's speed, this results in a severe delay until hardware acceleration becomes available.

A new approach to mitigate the long process of synthesis and implementation of FPGA designs while keeping the flexibility is presented in [8]. In this paper, methods to manipulate and generate partial bitfiles have been analyzed. The functional meaning of bitfile code segments has been made clear, thus creating access to direct manipulation of bitfiles. Using these methods, bitfile manipulation and consequently bitfile creation without the need to go through most of the steps of implementation can be achieved. By using a library of pre-synthesized micro modules, synthesis could also be foregone. Ultimately, a concept for run-time adaptation of an FPGA configuration based on a network grid with variable field sizes is proposed. Another very important aspect with regard to high reconfiguration times for FPGAs is partial reconfiguration, so that only a specific area can be reconfigured while the rest of the FPGA can continue working. Using the methods described in [9] in combination with partial reconfiguration could effectively cut the overall preparation time of an FPGA-based accelerator while keeping maximum adaptability. It is our goal to reduce the delay until which an accelerator becomes available to levels far below that of Warp [7] through these techniques.

III. Concept

Our approach is based on a proven run-time reconfigurable processor design, i.e., a reconfigurable fabric is tightly coupled to the pipeline of the processor and can be used to configure (one or more) accelerators [10]. Acceleration using the reconfigurable fabric is achieved from the CPU pipeline by employing *Special Instruction*s (SIs). An SI invokes execution of a microprogram on the SI Execution Controller, which takes care of data transfers between accelerators, accelerator execution and writeback of results to the CPU register file, i.e., an SI is implemented by a microprogram that utilizes one or more accelerators. For the pipeline, an SI is a multi-cycle instruction just like, e.g., integer division. Figure 1 shows how existing approaches utilize accelerators (either they are presynthesized offline for specific program parts or functionally equivalent software is executed) and the extensions for the new *Automatic SI* (Auto-SI) approach. Auto-SI detects compute-intensive program parts and builds accelerators flexibly at run time. To realize this in a dynamic and transparent way, four steps are necessary:

(A) Observe running instructions
(B) Prepare the accelerator
(C) Configure the reconfigurable fabric
(D) Use the accelerator

These steps take considerable time to perform. While no performance penalty is incurred due to Steps *(A)* to *(C)* being performed concurrently to software execution, to obtain an actual performance benefit in Step *(D)*, a selected kernel has to reoccur after the accelerator becomes available. Therefore, we focus on compute-intensive kernels as these have the highest probability of recurrence. There is a tradeoff between the number of times a kernel is executed and the performance gain, stemming from the overhead introduced by the Auto-SI steps. This tradeoff will be more closely analyzed in Section V.

A. Observe Running Instructions

A tracking unit has been added to the system to find loop kernels that are suitable for hardware acceleration. Its purpose is to observe the Program Counter (PC) and the corresponding instruction. Figure 2 shows a compute-intensive kernel that is tracked by Auto-SI for acceleration. When a branch instruction is detected and the branch is taken, the tracking unit stores the corresponding PC and tracks the following instructions.

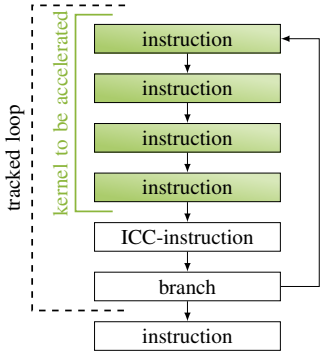

Figure 2: Compute-intensive Kernel, Tracked by Auto-SI for Acceleration.

Each tracked instruction is checked against a list of supported instructions, since not all types of instructions may be supported by an implementation of the Auto-SI concept. Unsupported instructions could stem from limitations like not having access to a certain pipeline register or a Floating Point Unit (FPU) / coprocessor. If an unsupported instruction is detected, the currently tracked kernel cannot be accelerated by Auto-SI. In case another branch is detected during tracking, the stored PC will be compared against the PC of the branch instruction that triggered the tracking process. If they have the same value, the instructions tracked up to this point constitute the kernel and are then further processed in the following step.

B. Prepare the Accelerator

Before an accelerator can be created, the detected kernel needs to be analyzed further to make sure it meets certain constraints. The main constraint is the interface between the CPU pipeline and the reconfigurable fabric. Through this interface, the accelerator receives inputs and also delivers its outputs, so a kernel accelerated using Auto-SI cannot require more inputs or outputs than this interface provides. Furthermore, the accelerator created by Auto-SI needs to fit within the resources available on the FPGA. Any kernel that fulfills all constraints can be accelerated. In order to verify the constraints, a Data Flow Graph (DFG) is extracted from the kernel that was tracked in the previous step. The nodes from the DFG are assigned to timeslices of a schedule according to a defined scheduling policy like ASAP or ALAP to estimate the critical path of the accelerator that is built for the kernel. Multiple nodes can be assigned to a single timeslice, so that Instruction Level Parallelism (ILP) can also be exploited.

Using the resulting DFG, we obtain a bitfile that implements the corresponding accelerator. Currently, this is achieved by providing a library of bitfiles and corresponding DFGs, both stored in memory, i.e., we try to find the extracted DFG within the provided DFGs. In case we find a matching DFG, a bitfile for an accelerator that implements the functionality of the tracked kernel is available. In future works, this method will be refined to avoid the main shortcoming of a fixed library, which can only provide a limited number of accelerators and consequently reduces the applicability of Auto-SI. Using methods described in [9, 11], the bitfile will be created during run time while forgoing many of the computing and memory intensive steps usually required in bitfile generation by employing a library of "micro-bitstreams", each replicating a single instruction. During run time, these will be assembled using the extracted DFG to produce any accelerator required.

The following steps are independent of the method of obtaining a bitfile. The result of this step is either a corresponding accelerator bitfile or the determination that the kernel does not meet all constraints and cannot be accelerated.

C. Configure the Reconfigurable Fabric

Once a bitfile for an accelerator that implements the functionality of the tracked kernel has been found, the accelerator will be configured in the reconfigurable fabric. Reconfiguring an accelerator is performed in parallel to the execution of the application. This is the most time-consuming step of the entire process. Reconfiguring an accelerator can require several thousand CPU cycles (depending on the CPU frequency, bitstream size and reconfiguration bandwidth), as will be discussed in Section V.

D. Use the Accelerator

As soon as the hardware accelerator is ready for use, the kernel will be accelerated in hardware. This requires manipulating the stream of instructions the CPU pipeline processes (to execute the newly available hardware accelerator instead of the original kernel code). When the beginning of the software kernel is detected via the stored PC of the branch instruction, an SI is inserted in the decode stage after the branch instruction and the PC is set to the instruction following the critical kernel, which is the branch instruction itself. Because the SI is a multi-cycle instruction, the pipeline holds the PC preventing new instructions from entering the pipeline, but allows instructions that are already in the pipeline to retire correctly. By using an SI, the call to the accelerator can make use of existing pipeline mechanisms like register forwarding or handling of multi-cycle instructions. The SI contains some of the required information to call the accelerator, the remainder being stored in dedicated

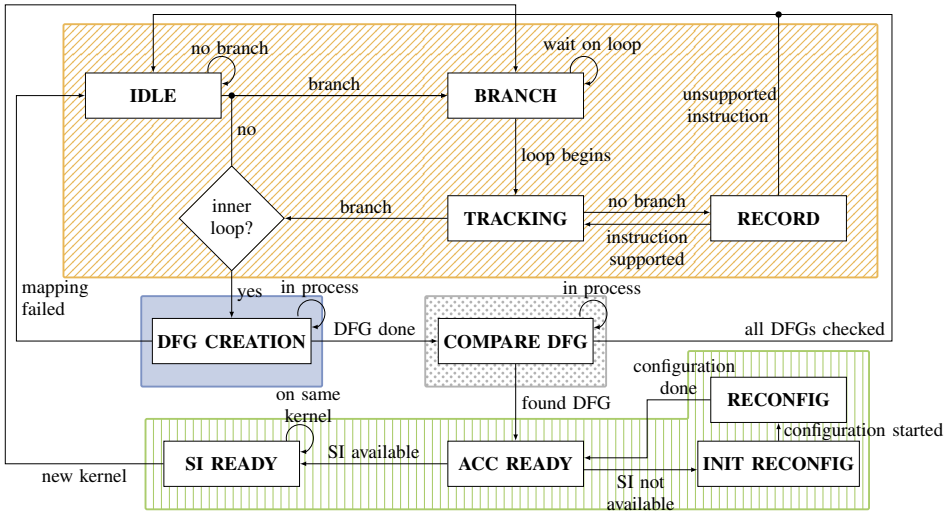

Figure 3: Process of the Auto-SI Approach

registers. The input values for the kernel are passed directly from the pipeline to the accelerator. As soon as the results are available, the pipeline collects the data and writes it back to the corresponding CPU registers. Afterwards, the branch instruction of the kernel enters the pipeline. If the branch is taken, this entire step is repeated. If it is not taken, the application continues executing the software that follows the kernel.

IV. Auto-SI Approach

Figure 4 summarizes the main steps of the Auto-SI approach and the result of each step that was introduced in the previous section. In this section, the finite state machine (FSM) that controls these steps is described in more detail.

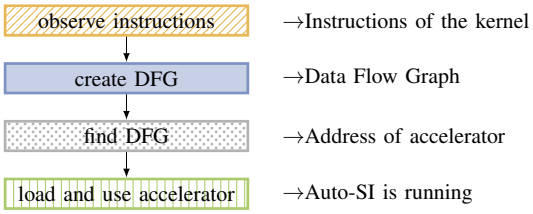

Figure 4: Auto-SI Flow and Output

The FSM is shown in Figure 3 and the corresponding states are marked with the same colors and patterns.

Initially, the Auto-SI FSM is in standby mode (state `IDLE`) and waits for branch instructions. Once a branch occurs, the current PC is stored and the FSM (state `BRANCH`) waits until the instructions of the branch target have entered the pipeline. From then on, each supported instruction of the potential kernel is tracked (state `TRACKING`). If there is an unsupported instruction, the FSM cancels the tracking and returns to the standby mode. In case another branch occurs while tracking, the FSM checks whether the branch corresponds to the previously stored branch. If the branch does not match to the previous one, the tracked data is discarded and the FSM will start tracking the next potential kernel. In case the branch matches the previously stored one, the kernel is suitable for Auto-SI and the creation of the DFG for this kernel starts (state `DFG CREATION`). The DFG creation might fail due to kernel-level constraints. In this case, the FSM returns to the standby mode, waiting for the next branch to occur. Once the DFG is successfully generated, it will be compared with the DFGs of available accelerators (state `COMPARE DFG`). If a suitable accelerator exists, it is first checked if the accelerator is already configured and usable (state `ACC READY`). If it is necessary to configure the accelerator, the reconfiguration is initiated (state `INIT RECONFIG`) and performed (state `RECONFIG`). Otherwise, the accelerator is ready for use (state `SI READY`) and from now on, the kernel can be executed in hardware. As soon as execution leaves the kernel, the FSM returns to the standby mode and observes the running application again.

V. Experimental Evaluation

We evaluated this work on a reconfigurable processor called *i*-Core [12] that we extended for the Auto-SI approach. The implementation is based on the Gaisler LEON3 SoC, with a SPARC V8 in-order microarchitecture. It was synthesized to a Xilinx Virtex-7 FPGA (XC7VX485T-2FFG1761C). The 7-stage pipeline of the LEON3 processor was extended into a reconfigurable processor core: the execute stage allows stalling the pipeline and passing operands to the fabric. An SI is implemented as a microprogram that is executed by the so-called SI Execution Unit. The protocol of executing SIs on the CPU pipeline is similar to other multi-cycle instructions like division, and can directly access register operands (see [10] for more details). The reconfigurable fabric is divided into equally-sized accelerator containers. Initiating a specific configuration is done by Auto-SI using the memory-mapped interface of the reconfiguration unit called CmdQueue (see [13] for more details).

Figure 5 shows the structure of the whole *i*-Core, its reconfigurable fabric and the added Auto-SI module. This module comprised the state machine presented in Section IV and an AHB interface. The accelerators and the according DFGs are both read from memory by using the AHB. In the following sections, the benefits of the proposed design are evaluated.

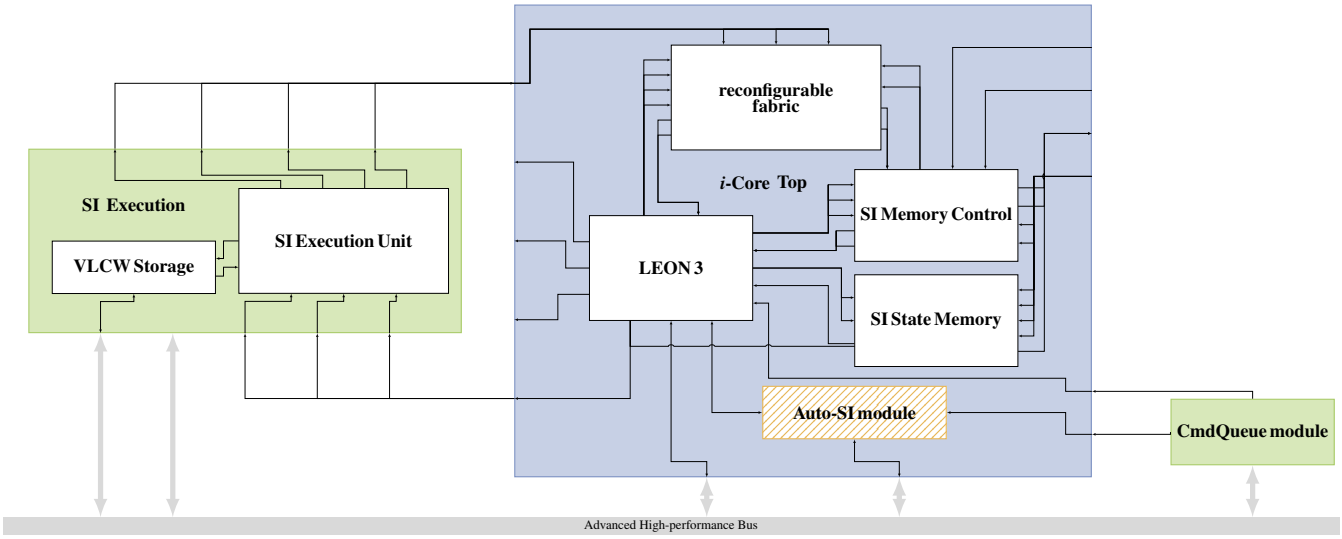

Figure 5: *i*-Core with Auto-SI Extension

A. FPGA Resources

Figure 6 shows the additionally used resources for the Auto-SI implementation compared to resources utilized by *i*-Core. The Auto-SI implementation requires less than 1% of the resources of the used Virtex-7. Compared with to the original *i*-Core, 8.5% more LUTs are needed to extend it with the Auto-SI approach. On average, 22% more resources are used in contrast to the original *i*-Core. The utilization of resources is not correlated to the amount of accelerated instructions or the size of the accelerator.

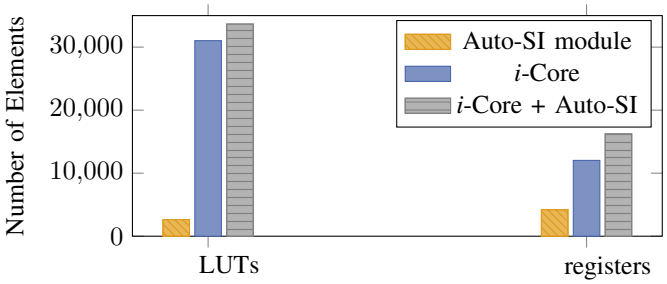

Figure 6: Resource Utilization of the Auto-SI Approach

B. Delay Analysis

In the following, we will estimate the maximum delays incurred by the Auto-SI approach analytically. Afterwards, we will evaluate the tradeoff between the obtainable speedups and the delay incurred by Auto-SI. Table I shows the various factors that have been considered to calculate the total delay.

The access time $t_{\text{RAM\_access}}$ depends on the utilization of the AHB and is estimated through measurements. The access time of the storage also depends on run-time conditions and is estimated. As introduced in Section IV, the following four steps are required to obtain acceleration:

1) Track instructions D_{track}
2) Create DFG D_{DFG}
3) Find DFG D_{match}
4) Load accelerator D_{reconfig}

factor	symbol
amount of instructions within the kernel	n
length of the loop	m
amount of prepared bitfiles	k
length of the DFG i	L_i
utilization of the AHB	$load_{\text{AHB}}$
access time storage	$t_{\text{RAM\_access}}$
access time AHB	$t_{\text{AHB\_wait}}$

Table I: Determining Factors

The sum of all these execution times is the overall maximum delay D in cycles. For each instruction within the kernel the tracking algorithm needs one clock cycle:

$$D_{\text{track}} = n \tag{1}$$

Also the delay of the DFG generation depends on the amount of instructions within the tracked kernel:

$$D_{\text{DFG}} = \frac{1}{2} \cdot (n^2 + 9n + 4) \tag{2}$$

The delay of the bitfile matching algorithm depends on the length of the created DFG and the amount of available bitfiles k. To calculate D_{match}, it is assumed that the requested bitfile is the last loaded one. In addition, the access time of the storage has to be taken into account. The DDR3-memory of the Virtex-7 board is only able to load eight data words with the length of 32 bits within one consecutive burst. Due to this, after 32 bytes an additional waiting time $t_{\text{RAM\_wait}}$ has to be added. Furthermore, incremental AHB bursts are not able to exceed address ranges of 1kB [14]. The AHB interface uses the maximum length of incremental bursts (8 transfers). Therefore, 32 bursts are needed to transfer 1kB and 32 wait states have to be added. The experiments deliver $t_{\text{RAM\_wait}} \approx 40$ clock cycles. Overall, the data volume for all DFGs is $2 \cdot \sum_{i=0}^{k} L_i \cdot 4$ Bytes and the corresponding delay D_{match} for the bitfile matching algorithm is:

$$D_{\text{match}} = 12 \cdot \sum_{i=0}^{k} L_i \tag{3}$$

The delay of the reconfiguration is influenced by the utilization of the AHB, the size s_{bitfile} of the loaded bitfile, the clock frequency f and the transmission rate r:

$$D_{\text{reconfig}} = t_{\text{AHB\_wait}} + t_{\text{reconfig}} \tag{4}$$

The reconfiguration itself corresponds to $t_{\text{reconfig}} = \frac{s_{\text{bitfile}}}{r} \cdot f$ and is reviewed in more detail in [13].

C. Experimental Results

To verify the analysis of the Auto-SI delays, we evaluate our hardware prototype with several test kernels consisting of a varying amount of sequential ADD instructions.

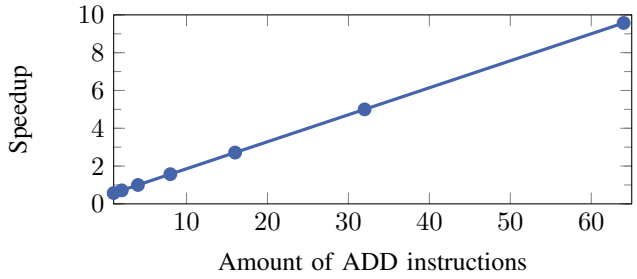

Figure 7: Acceleration – Sequential ADD Instructions

Figure 7 shows the possible speedup for these kernels according to the amount of instructions. The speedup reflects the relation between the execution time of accelerated kernel and the execution time of the software kernel. The more instructions within a kernel are accelerated, the more speedup is achievable. A maximum limit for the amount of instructions is only given by the limitation of the size of the reconfigurable area of the *i*-Core.

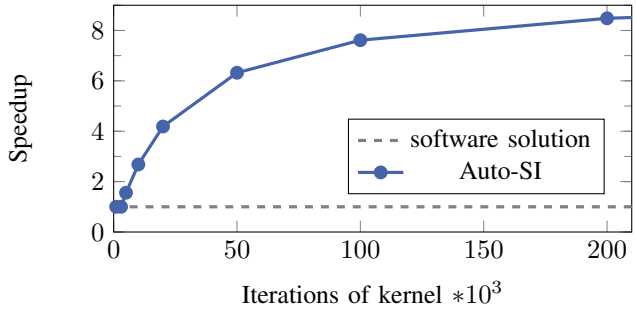

Figure 8: Acceleration – Kernel of 64 Sequential ADD Instructions

The actual achieved performance gain is a result of the speedup obtained from the accelerator and the number of iterations of the kernel. In addition, the reconfiguration delay has to be taken into account, the results for a kernel of 64 sequential ADD instructions is given in Figure 8. In our experiments, the accelerator was loaded and ready after 1591 loop iterations, after this period a speedup by the accelerator is given. Note that this assumes that the accelerator should be detected and used within a single execution of the loop. If the accelerator was loaded in a previous execution and the loop is reentered, it will be used immediately after the Auto-SI unit found the according DFG in the library. In our experiments, this process requires 14 loop iterations of the kernel.

VI. Conclusion and Future Work

In this paper, we proposed an automatic loop detection and hardware acceleration approach for an adaptive reconfigurable processor. The Auto-SI approach is implemented and integrated into a SPARC V8 microarchitecture. In the first experiments, a transparent speedup up to the factor $9.5\times$ is demonstrated on sequences of ADD instructions, with an additional overhead of 22% resource consumption. The relation shown between overhead and speedup $(43 : 1)$ of these first results is very promising. In the future, the design will be enhanced to also handle outer loops and more complex instruction structures. One further extension is the generation of more generic accelerators that can be used for different DFGs, e.g., to accelerate floating-point operations [15].

Acknowledgment

This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre “Invasive Computing” (SFB/TR89).

References

[1] J. Henkel, A. Herkersdorf, L. Bauer *et al.*, “Invasive Manycore Architectures,” in *Proceedings of the 17th Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan. 2012, pp. 193–200.

[2] G. Ansaloni, P. Bonzini, and L. Pozzi, “Design and Architectural Exploration of Expression-Grained Reconfigurable Arrays,” in *Symposium on Application Specific Processors (SASP)*, 2008, pp. 26–33.

[3] L. Chen, J. Tarango, T. Mitra, and P. Brisk, “A Just-in-Time Customizable Processor,” in *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, November, 2013, pp. 524–531.

[4] A. Sharifian, S. Kumar, A. Guha, and A. Shriraman, “CHAINSAW: Von-Neumann Accelerators To Leverage Fused Instruction Chains,” in *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, October 2016, pp. 1–14.

[5] M. A. Watkins, T. Nowatzki, and A. Carno, “Software Transparent Dynamic Binary Translation for Coarse-Grain Reconfigurable Architectures,” in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 138–150.

[6] A. C. S. Beck, M. B. Rutzig, and L. Carro, “A Transparent and Adaptive Reconfigurable System,” *Microprocessors and Microsystems*, vol. 38, no. 5, pp. 509–524, 2014.

[7] R. Lysecky, G. Stitt, and F. Vahid, “Warp Processors,” *ACM Transactions on Design Automation of Electronic Systems*, vol. 11, no. 3, pp. 659–681, July 2006.

[8] L. Braun and J. Becker, “Two Dimensional Dynamic Multigrained Reconfigurable Hardware,” in *VLSI 2010 Annual Symposium: Selected papers*, Jun. 2011, vol. 95, pp. 303–318.

[9] L. Braun, “Methoden zur Erstellung eines laufzeitadaptiven und zweidimensional rekonfigurierbaren Systems,” Ph.D. dissertation, ITIV, Karlsruher Institut für Technologie, 2013.

[10] L. Bauer, M. Shafique, and J. Henkel, “A computation-and communication-infrastructure for modular special instructions in a dynamically reconfigurable processor,” in *Int. Conf. on Field Programmable Logic and Applications*. IEEE, 2008, pp. 203–208.

[11] R. K. Soni, N. Steiner, and M. French, “Open-source bitstream generation,” in *IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2013, pp. 105–112.

[12] J. Henkel, L. Bauer, M. Hübner, and A. Grudnitsky, “i-Core: A run-time adaptive processor for embedded multi-core systems,” in *Int. Conf. on Engineering of Reconfig. Syst. and Algorithms*, 2011.

[13] M. Damschen, L. Bauer, and J. Henkel, “Timing Analysis of Tasks on Runtime Reconfigurable Processors,” *IEEE Trans. on Very Large Scale Integration Syst.*, vol. 25, no. 1, pp. 294–307, Jun. 2016.

[14] *AMBA Specification*, Revision 2.0 ed., Arm Limited, 1999. [Online]. Available: http://soc.eecs.yuntech.edu.tw/Course/SoC/doc/amba.pdf

[15] L. Bauer, A. Grudnitsky, M. Damschen *et al.*, “Floating Point Acceleration for Stream Processing Applications in Dynamically Reconfigurable Processors,” in *IEEE Symposium on Embedded Systems for Real-time Multimedia (ESTIMedia)*, Oct. 2015.