Efficient Resource Utilization for an Extensible Processor through Dynamic Instruction Set Adaptation

Lars Bauer, Muhammad Shafique Student Members IEEE, and Jörg Henkel Senior Member IEEE

Abstract—State-of-the-art ASIPs (Application Specific Instruction set Processors) allow the designer to define individual pre-fabrication customizations, thus improving the degree of specialization towards the actual application requirements, e.g. the computational hot spots. However, only a subset of hot spots can be targeted to keep the ASIP within a reasonable size. We propose a modular Special Instruction composition with multiple implementation possibilities per Special Instruction, compile-time embedded instructions to trigger a run-time adaptation of the Instruction Set, and a run-time system that dynamically selects an appropriate variation of the Instruction Set, i.e. a situation-dependent beneficial implementation for each Special Instruction. We thereby achieve an up to 3.0x (avg. 1.4x) better efficiency of resource usage compared to current state-of-the-art ASIPs resulting in a 3.1x (avg. 1.4x) improved application performance (compared to a GPP up to 25.7x and avg. 17.6x).

Index Terms—extensible processor, ASIP, reconfigurable architecture, run-time adaptation, modular special instructions, RISPP, Rotating Instruction Set Processing Platform

I. INTRODUCTION AND RELATED WORK

A general overview of the benefits and challenges of ASIPs is given in [1], [2]. Due to vendors like Tensilica [3], ARC [4], CoWare [5], TargetCompiler [6] etc. the designer can now implement a specific instruction set that is tailor-made for a certain set of applications. Typically, these suites come with a whole set of retargetable tools such that the code can be generated conveniently for a specific ASIP. As the instruction set definition requires both application and hardware architecture expertise, a major research effort was spent in design space exploration [7] and automatically detecting and generating so called Special Instructions (SIs) from the application code [8]. A library of reusable functions is used in [9], whereas in [10], [11] the authors describe methods to generate SIs through matching profiling patterns. The authors in [12] investigate local memories in the functional units, which are then exploited by SIs. An automated, compiler-directed system for synthesizing accelerators for multiple loops (multifunction loop accelerators) is presented in [13]. The authors in [14] present an approach to exploit similarities in data paths by finding the longest common subsequence of multiple data paths to increase their reusability.

Present age complex applications, e.g. from the multimedia domain, consist of multiple hot spots each requiring various different hardware accelerators to achieve the desired performance. We have analyzed the relative computational time for the major processing functions in the H.324 [15] video conferencing application (see Fig. 1). H.324 consists of video (H.264 [16]), audio (G.723 [17]) codecs and V.80 protocol that specifies how modems should handle streaming audio and video data. Many of the processing functions require diverse hardware accelerators to achieve a certain overall speedup, therefore resulting in large area requirements.
FPGA-like reconfigurable fabric on which SIs are dynamically loaded during run time. Our adaptive extensible processor with its fine-grained reconfigurable fabric extends the previously presented approaches by its novel vision of modular SIs and a run-time system that uses the modular SIs to enable an adaptive and efficient utilization of the available hardware without statically predetermined reconfiguration decisions.

Chimaera [25] couples a small and fast FPGA-like Reconfigurable Array with a superscalar processor. The processor is stalled while the array is reconfigured and in the case of large working sets (i.e. many SIs within a loop), the problem of thrashing in the configuration array is reported (i.e. frequent reconfigurations within each loop iteration). Our approach instead neither stalls execution while reconfiguring nor is thrashing an observed problem. In our RISPP approach we offer multiple implementations of each SI to provide different performance-area trade-offs. Depending on the number of SIs required in one loop, smaller or bigger implementations are automatically selected.

XiRisc couples a VLIW processor with a reconfigurable gate array. The configuration is selected out of four different contexts and reconfiguration between them can be done in a single cycle. These multiple contexts are beneficial if small applications fit into them. In [26] the fastest reported speedup (13.5x) is achieved for DES and the only context reloading happened when the application was started. However, in [27] a relevant MPEG-2 encoder is used for benchmarking. Here, run-time reconfiguration is required (as the accelerators no longer fit into the available contexts) and the achieved speedup reduced to 5x compared to the corresponding processor without reconfigurable hardware (i.e. GPP).

The Molen Processor couples a reconfigurable processor to a base processor via a dual-port register file and an arbiter for shared memory [28]. The application binary is extended to include instructions that predetermine the reconfigurations and the usage of the reconfigurable coprocessor. The OneChip98 project [29] instead uses a tighter coupling of the reconfigurable hardware as Reconfigurable Functional Units (RFUs) within the core pipeline. As their speedup is mainly obtained from streaming applications, they allow their RFUs to access the main memory, while the core pipeline continues executing. Both approaches offer one implementation per SI. Our approach instead envisions modular SI implementations that offers different alternatives and thus allows flexibility and an efficient utilization of the reconfigurable hardware.

The Warp Processor [30] automatically detects hot spots while the application executes. Then, custom logic for the SIs is generated at run time through on-chip micro-CAD tools and the binary of the executing program is patched to execute them. However, the online synthesis incurs a non-negligible overhead and therefore the authors concentrate on scenarios where one application is executing for a rather long time without significant variation of the execution pattern. In these scenarios, only one online synthesis is required (i.e. when the application starts executing) and thus the initial performance degradation accumulates over time. However, adaptation to frequently changing requirements cannot be addressed by this scheme. Our modular SIs instead uses pre-synthesized data paths with predetermined possibilities how they can be connected to realize compile-time determined SIs with different trade-offs.

All the above discussed approaches potentially increase the utilization of the available hardware resources by reconfiguring parts of the hardware to match the current requirements of the application (i.e. the currently executing hot spots). However, due to the reconfiguration time the utilization of the reconfigurable area may often be sub-optimal. The reconfiguration time for coarse-grained reconfigurable architectures is shorter, but they cannot implement state machines or bit manipulations efficiently. Therefore, these architectures are mainly beneficial for applications that target data-flow processing, while ASIPs may additionally accelerate bit manipulations or hot spots with embedded control-flow.

In order to overcome these shortcomings, we have designed a novel run-time adaptive extensible processor RISPP (Rotating Instruction Set Processing Platform). We thereby offer a platform to use the available hardware resources efficiently by implementing modular SIs in a fine-grained partial reconfigurable fabric. Compared to state-of-the-art reconfigurable architectures, we reduce the time until a certain SI can use the reconfigurable accelerators by the ability to utilize elementary reconfigurable data paths without the constraint to wait for the complete reconfiguration of that SI (i.e. exploiting full parallelism). We achieve these goals by a modular SI composition (i.e. an SI is composed of elementary data paths as a connected module), which is mainly driven by the idea of a high degree of reusability of data path elements. While reusable data path elements are used in ASIP designs as well, state-of-the-art reconfigurable architectures did not consider this possibility up to now. State-of-the-art ASIPs instead did not consider run-time reconfiguration (to e.g. reconfigure from a clipping function to a bit manipulation) of their executing units. Our goal is to use the available area efficiently and we will compare our efficiency and hardware utilization against ASIPs in Section IV, as ASIPs already provide proper hardware utilization due to their reusable data path elements. This does not mean that our concept is limited or optimized for area-constrained devices, but it means that we aim to achieve a high performance for a significantly reduced footprint (due to the efficient area usage). In the case of devices with high amount of available area, the remaining area can be used to implement e.g. a second processor (multiprocessor), a cache, etc. and thereby increase the performance further.

Our contributions are:

- a novel extensible processor with modular Special Instructions that allows run-time adaptation and achieves a higher efficiency of resource usage and better performance compared to state-of-the-art
- an architecture description, run-time system, and hardware prototype to realize and evaluate our novel platform
- a detailed analysis and exploration of the resource utilization design space for multimedia applications with a focus on video encoders, comparing with different ASIP implementations while showing their benefits and limitations

The rest of the paper is organized as follows: In Section II we analyze the resource utilization of ASIPs. Section III pro-
vides the detailed description of our architecture with a comprehensive explanation of our *modular Special Instruction* composition, early reconfiguration, run-time system, and hardware prototype. In Section IV we explain the performed comparison and the used benchmark application before presenting a detailed evaluation, comparison, and discussion of our proposed approach. We conclude our work in Section V.

II. Mmotivational Case Study

We have analyzed the detailed execution behavior of the H.264 video encoder for a state-of-the-art *ASIP* by considering the major computational hot spots and by implementing accelerating data paths for hardware execution. This case study is meant to illustrate a problem when *ASIPs* face large-sized real-world applications. Typical benchmarks for *ASIPs* comprise MiBench [31] and MediaBench [17]. They are only considering computational hot spots like *DCT, SAD, VLC, FIR, etc.* (these typical hot spots are components of the H.264 video encoder). For applications with few hot spots, *ASIPs* seem to be a good solution. However, if we consider a big application like the H.264 video encoder, then the hardware footprint for an *ASIP* will grow significantly, as we will see. Therefore, an application like the complete H.264 encoder is typically not considered as benchmark (however, sometimes parts like *DCT* or *SAD* are extracted and accelerated stand-alone).

Table I: Implemented Special Instructions and data paths for the major functional components of H.264 Video Encoder

<table>
<thead>
<tr>
<th>Functional Component</th>
<th>Special Instruction</th>
<th>Accelerating Data Paths</th>
</tr>
</thead>
<tbody>
<tr>
<td>Motion Estimation (ME)</td>
<td>SAD</td>
<td>SAD,16</td>
</tr>
<tr>
<td>Motion Compensation</td>
<td>SATD</td>
<td>QSub, HT_4, Repack, SATD</td>
</tr>
<tr>
<td>Intra Prediction (IPred)</td>
<td>MC_Hz.4</td>
<td>PointFilter, BytePack, Clip3</td>
</tr>
<tr>
<td>Transform</td>
<td>I Pred HDC</td>
<td>PackLBytes, CollapseAdd</td>
</tr>
<tr>
<td></td>
<td>I Pred VDC</td>
<td>CollapseAdd</td>
</tr>
<tr>
<td></td>
<td>(I16HT)</td>
<td>DCT_4, Repack, (QSub)</td>
</tr>
<tr>
<td></td>
<td>(I4HT) x4</td>
<td>HT_2</td>
</tr>
<tr>
<td></td>
<td>(I4HT) x4 HT_4, Repack</td>
<td></td>
</tr>
<tr>
<td>Loop Filter (LF)</td>
<td>LF BS4</td>
<td>Cond, LF_4</td>
</tr>
</tbody>
</table>

As the standard implementation is inefficient in terms of memory and processing it is normally not used for performance testing. Therefore, we have used an optimized H.264 encoder application that serves as our benchmark application. The optimizations comprise: selecting the *Baseline* profile tools, data structure improvement, using an optimized *Motion Estimation* scheme, and improving the data flow (the details of these application optimizations are given in Section IV.B). Afterwards, we profiled the application and detected the computational hot spots. For each hot spot, we have designed and implemented several *Special Instructions* (SIs, composed of hardware accelerators). Table I shows the corresponding SIs and the accelerating data paths that we have implemented for the major functional components of the H.264 encoder. Multiple SIs may share a data path (e.g. the *QSub* and *Repack* data paths for *SATD* and *DCT*). Depending on the available hardware resources (and thus the number of data paths that can be implemented) some of the SIs in Table I might not be accelerated by hardware at all, whereas in other cases some SIs were implemented using multiple instances of one data path to exploit the inherent parallelism (e.g. *SAD* may be implemented with one or two instances of the *SAD_16* data path).

Fig. 2 shows the execution time (bars) of the H.264 video encoder for different quantities of deployed data paths. As the data paths are of similar sizes (between 5,599 and 8,799 gate equivalents, see Section III.C), the amount of used data paths indicates the area extensions of the *ASIP*. Although our RISPP concept and its implementation do not rely on similar sized data paths, it simplifies discussion and comparison. The selection of additional data paths was done optimally (concerning the application execution time) for the specific input video sequence. The optimal selection of data paths for this particular test video sequence will give the maximum benefit to *ASIP*, even though in real world scenarios the pattern of the input video typically cannot be known beforehand.

Table II: Selected Data Paths for the major functional components of H.264 Video Encoder

<table>
<thead>
<tr>
<th>Number of Available Data Paths</th>
<th>Selected Data Paths</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SAD 16</td>
</tr>
<tr>
<td>2</td>
<td>SAD 16, HT_4</td>
</tr>
<tr>
<td>3</td>
<td>SAD 16, HT_4, Cond</td>
</tr>
<tr>
<td>4</td>
<td>SAD 16, HT_4, Repack, QSub</td>
</tr>
<tr>
<td>5</td>
<td>SAD 16, HT_4, Repack, QSub, SATD</td>
</tr>
<tr>
<td>6</td>
<td>SAD 16, HT_4, Repack, QSub, SATD, DCT_4</td>
</tr>
<tr>
<td>7</td>
<td>SAD 16, HT_4, Repack, QSub, SATD, DCT_4, Cond</td>
</tr>
</tbody>
</table>

After determining the application execution time in Fig. 2, we have analyzed the efficiency of the data path usages (equation 1), i.e. the speedup per available data path, where the speedup is relative to the execution time of a *GPP* (i.e. a MIPS core without any hardware accelerators; requiring 7.4 billion cycles to encode 140 video frames with our H.264 encoder).

Fig. 2: Analyzing the execution time and the resource usage efficiency using different area deployments when processing 140 video frames.

Table II shows the selected data paths for the first seven readings in Fig. 2. Note that some of the selected data paths are reused to implement different SIs, e.g. *HT_4* is used by *SATD* and *(I)*HT_4x4, as shown in Table I. One interesting situation can be seen when moving from 3 to 4 data paths in Table II. While in each other increment step the previously selected data paths are extended by an additional data path, in this step the previously selected *Cond* data path is discarded and two new data paths are selected instead. This is because *Repack* and *QSub* are required together to achieve a noticeable performance improvement of *SATD*, whereas *LF BS4* can be accelerated even if only *Cond* is available.

After determining the application execution time in Fig. 2, we have analyzed the efficiency of the data path usages (equation 1), i.e. the speedup per available data path, where the speedup is relative to the execution time of a GPP (i.e. a MIPS core without any hardware accelerators; requiring 7.4 billion cycles to encode 140 video frames with our H.264 encoder).
Efficiency := \frac{\text{Speedup}}{\# \text{Data Paths}} * \frac{\text{without data paths}}{\text{relative to the execution time}} (1)

We allow an SI to be implemented with a subset of its required data paths, e.g. the SATD SI in Table I can be accelerated (with a smaller speedup) even if only the HT_4 data path is available (the remaining computation is performed without hardware accelerators). The speedup for the first added data path is already 2.4x, which results in a good efficiency (see the efficiency-line in Fig. 2). However, to achieve a better performance, more data paths have to be added. This leads to a significant efficiency decrease, as e.g. doubling the number of data paths does not double the speedup. This is because only few SIs can be accelerated using such a small amount of data paths. To obtain a good compromise between execution time and efficiency in Fig. 2, about 10 data paths have to be added. After adding 13 data paths, the efficiency is again decreasing, as the speedup is limited by the sequential part of the application. The result of this analysis is that up to a certain quantity of data paths the resource utilization of an ASIP is inefficient, thus limiting the potential speedup. A large amount of data paths needs to be added to match the required performance. This is a significant problem considering large applications like H.324 in Fig. 1 (the H.264 encoder is one component of H.324) or even multiple applications in a multitasking environment.

This definition is based on the observation that in the best case each SI execution in a timeframe could make use of all available data paths, which then corresponds to 100% utilization according to our definition. The bars in Fig.3 show which data paths were actually used per timeframe (for clarity it is not shown how often they were used). The drawn-through line corresponds to the data path utilization and the dashed line shows the average data path utilization for the whole execution. The maximum number of available data paths (six in this example) is only used in timeframe 5. In this timeframe, the processing flow changes from Motion Estimation (using five data paths) to Encoding Engine (using three data paths) which shows that not all six data paths are used at the same time. It can be seen that the Repack, QSub, and HT_4 data paths are used for both hot spots, thereby increasing the average utilization and thus the overall efficiency. Except these three, all other data paths are dedicated to a specific SI and therefore they are not utilized efficiently. In timeframes 16 to 26 (execution of Loop Filter) not even one of the available data paths can be used, which results in a disadvantageous average utilization of 17.7%. It is due to the fact that for six available data paths no accelerator for the Loop Filter SI was selected. Instead, to achieve the best overall performance, all six available data paths were given to the Motion Estimation and Encoding Engine. After the Loop Filter finished execution, the Motion Estimation for the next incoming video frame starts in timeframe 27.

If we can improve the efficiency of the hardware usage, then we can achieve a good performance with less data paths. Thus we would save area (and costs, static power consumption, etc.) or we could use the area for other components, e.g. Caches.

III. ARCHITECTURE DESCRIPTION

Our adaptive extensible processor RISPP (Rotating Instruction Set Processing Platform) achieves its efficiency by using the available hardware resources in a time-multiplexed manner by reconfiguring parts of the hardware to contain only those data paths that are required at a certain point in time during execution. State-of-the-art reconfigurable architectures instead offer monolithic Special Instruction (SI) as their basic reconfigurable blocks. We gain extra performance by offering reusable data paths as our elementary reconfigurable units to constitute an SI (as we will see later). Compared to state-of-the-art reconfigurable computing approaches our approach leads to two major advantages:

- We can reuse the data paths to share them between different SIs (e.g. Repack and QSub as shown in Table I).
- We can use the data paths as soon as they are reconfigured. Therefore, we do not need to wait until the full implementation of an SI is loaded. Instead of this an SI can be gradually upgraded during run time.
In our architecture we extend a typical pipeline (the so-called core pipeline) by a partial reconfigurable hardware and a Run-Time Manager. For evaluation, we are currently working with MIPS and SPARC but we are not limited to a specific core pipeline. The partial reconfigurable hardware is tightly coupled to the core pipeline as shown in Fig. 4. This means that the reconﬁgurable hardware can obtain input data directly from the register file. We have extended the register file to four read ports to transfer sufﬁcient data to the SI implementations. The partial reconﬁgurable hardware is additionally connected to a data-memory access-unit that accesses (via an arbiter) the data memory hierarchy. This allows streaming operations, e.g., the base address and the stride (i.e., the access pattern) of an array access can be provided by the register file and the SI can then accesses the corresponding memory addresses. Such an SI requires multiple cycles to complete. We stall the core pipeline while its execution to avoid conﬂicts between memory accesses from the core pipeline and the SI.

The potential beneﬁt of allowing the core pipeline to execute in parallel was investigated in [29]. Nine different memory consistency problems were reported and had to be solved by an intelligent memory controller while at the same time only the JPEG application took advantage of the overlapping execution. We therefore followed the conclusion and recommendation from [29] and decided to stall the execution of the core pipeline as long as an SI executes. This leads to a simpliﬁed arbiter, as there is only one possible inconsistency problem left which can be handled efﬁciently: if the ﬁrst activity of the SI is a memory access and the instruction that was directly preceding this SI was a load/store instruction then both memory accesses collide. However, as the load/store instruction was issued before the SI, it also has to access the memory ﬁrst. Thus, whenever a collision between a load/store instruction and an SI occurs, the access of the SI is delayed, until the load/store instruction ﬁnishes. Nevertheless, our general concept is orthogonal to stalling the core pipeline or executing in parallel.

**Fig. 5**: Composition of DCT_4x4 Special Instruction with the details of our reusable DCT/IDCT data path.

Fig. 5 shows the modular composition of the DCT_4x4 SI out of 3 elementary data paths. The computation takes eight 32-bit inputs and computes eight 32-bit outputs, using two provided memory ports (e.g., Tensilica [3] is offering for their cores [32]). The SIs and the data paths in this paper are manually developed for benchmarking purpose of our proposed architecture. A large effort has been spent to automatically detect SIs (for more details see previous works [10], [11]) but these techniques do not partition SIs into data paths. The automatic detection of SIs with connecting data paths (as shown in Fig. 5) is a research challenge and is beyond the scope of this paper. Conceptually, after partitioning SIs into data paths by e.g., graph partitioning algorithms, similar kind of data paths can be merged to give a more reusable data path (e.g., [14] presents an approach to exploit similarities in data paths by ﬁnding their longest common subsequence).

The internal assembly of the DCT_4 data path in Fig. 5 demonstrates how multiplexers increase the reusability by offering both the DCT and Inverse DCT butterfly. To compute the whole DCT on a 4 by 4 array (a so-called Sub-Block), eight executions of a DCT_4 data path are required. If the ASIP offers only one DCT_4 data path in hardware, then it has to use this data path eight times (in addition to the instances of QSub and Repack) to complete the DCT_4x4 SI. However, if four DCT_4 instances are available, each of them only needs to execute twice. The ASIP addresses this decision at design time and chooses data paths that are then made available statically. Traditional reconﬁgurable architectures choose one ﬁxed composition out of data paths at compile time and at run time they have to wait until the reconﬁguration is completed. The more data paths they choose for the SI implementation, the bigger is the reconﬁguration overhead. Our RISP architecture instead proposes modular SI compositions, as motivated in Fig. 5. We offer different implementation possibilities (prepared at compile time) for each SI, which employ different trade-offs between the amount of required accelerating data paths and the achieved performance. Our architecture can thereby upgrade the performance of an SI implementation gradually at run time. As soon as a e.g., a second DCT_4 data path ﬁnished reconﬁguration, it may be used to improve the performance of the corresponding SIs. Each SI additionally exists in one speciﬁc software implementation that does not need any accelerating data paths for execution. This implementation realizes the SI with the core pipeline, i.e., as the GPP would execute it.

In the case that the reconﬁgurable hardware does not support the hardware execution of a requested SI (either because the reconﬁguration is not ﬁnished yet or because the Run-Time Manager (see Section III.B) decided not to accelerate this SI with data paths) a trap for ‘unimplemented SIs’ is activated. The trap handler then determines which SI caused the trap to call a corresponding software implementation for it. To accelerate the trap handler, special hardware support is added to simplify the tasks of:

- determining which SI caused the trap
- determining the corresponding input parameters (saved in “temporary storage for sw-emul” in Fig. 4)
- managing the ‘write back’ to the expected registers

Although this special hardware support results in a noticeable reduction of the trap overhead, its area requirements in the hardware implementation are insigniﬁcant. This hardware support does not perform any kind of computation, but it extracts and moves data from one register to another. For instance, to determine which SI caused the trap, only the SI opcode has to be extracted from a ﬁxed position of the instruction word. This corresponds to a simple rewiring in the hardware implementation (e.g., extract and shift bits 28 to 24 towards bits 4 to 0). Without the hardware support the following steps would need to execute:
1. determine the address of the SI that caused the trap
2. load the corresponding instruction from memory
3. initialize a mask to extract the opcode-relevant bits
4. apply this mask to the loaded instruction word
5. shift the extracted opcode towards bit position 0

1. set trap mask to 0
2. issue array of hardware instructions
3. set trap flag
4. clear trap mask

DCT_4x4(Curr_4x4, Pred_4x4, Trans_4x4)
Input: Curr_4x4 : Current 4x4 sub-block
Pred_4x4 : Prediction 4x4 sub-block
Output: Trans_4x4 : 4x4 array of Transformed Coefficients

// DPs: Datapaths
IF (#QSub_DPs=0) and (#Repack_DPs=0) and (#DCT_4_DPs=0)
DCT_SWImpl_1(Curr_4x4, Pred_4x4, Trans_4x4);
ELSE IF (#QSub_DPs=0) and (#Repack_DPs=0) and (#DCT_4_DPs=1)
DCT_SWImpl_2(Curr_4x4, Pred_4x4, Trans_4x4);
ELSE IF (#QSub_DPs=0) and (#Repack_DPs=1) and (#DCT_4_DPs=1)
DCT_SWImpl_3(Curr_4x4, Pred_4x4, Trans_4x4);
...

Fig. 6: Pseudo-code for the trap handler that realizes the DCT_4x4 execution when not all data paths are available

(QSub, Repack, DCT_4x4)
Input: Curr_4x4 : Current 4x4 sub-block
Pred_4x4 : Prediction 4x4 sub-block
Trans_4x4 : 4x4 array of Transformed Coefficients
Output: Res0 : Residue Calculation in SOFTWARE
Res1 : Residue Calculation in HARDWARE

// Residue Calculation in SOFTWARE
// Vertical Transform in HARDWARE (results written to output array)
DCT_4x4(Trans_4x4, Curr_4x4, Pred_4x4)
Begin
For (i=0 to 3) {
(Res0, Res1) = VerticalTransform(Curr_4x4, Pred_4x4);
}

DCT_4x4 execution when not all data paths are available
time varying situations (e.g. depending on input data) that result in different SI execution frequencies. Fig. 8b shows the resulting execution pattern of FIs and SIs from Fig. 8a on a time scale. It shows that fine-tuning the \textit{Forecast Values} has a direct impact on the next iteration of the outer loop in Fig. 8a, when \( F_1 \) executes next time.

**Example for Control Flow Graph (Nodes are Base Blocks)**

At compile time, we profile the application and then use a graph analysis technique to determine points in the control flow graph that provide sufficient time to start the reconfiguration before the SI is deployed. In addition, the control flow from these points to the actual SI execution is examined to assure a high probability that the SI execution is actually reached. The specific details about automatically finding good points for the FIs are described in [34].

**B. Run-Time Architecture**

The normal (i.e. non-special) assembly instructions of the application are executed solely by the \textit{core pipeline} (i.e. without our extensions). Whenever an SI or FI is to be executed, the \textit{Run-Time Manager} is involved. The main tasks of the \textit{Run-Time Manager} are:

- Control the SI executions
- Fine-tune the \textit{Forecast Values} to adapt to changing / varying situations
- Determine data path re-loading decisions to implement the forecasted SIs

The integration of the \textit{Run-Time Manager} into the \textit{core pipeline} was shown in Fig. 4, whereas Fig. 9 gives internal details of the \textit{Run-Time Manager}, concentrating on the internal composition and their state transitions. The initial state (\textit{Decode}) triggers the execution of sub-state machines that are executed in parallel with the \textit{Decode} state. Note that a state does not necessarily correspond to a single clock cycle: some subsequent states are executed in the same cycle (or pipelined), whereas other states require multiple cycles.

The components of the \textit{Run-Time Manager} are triggered either by SIs or by FIs, as shown in Fig. 9. On the implementation side, the \textit{Run-Time Manager} can be partitioned into two parts: A synchronous part running in the clock domain of the \textit{core pipeline} and an asynchronous part that may run in a different clock domain. The synchronous part comprises fine-tuning the \textit{Forecast Values} and all parts that are triggered by SIs. These parts have to be synchronous to the \textit{core pipeline}, as they are tightly coupled to it (e.g. sending control signals or receiving status information). While adding the synchronous part to our hardware prototype, high effort was spent to make sure that the critical path of the \textit{core pipeline} is not affected. If an SI is about to execute, but the preceding fine-tuning is not finished yet, then the \textit{core pipeline} is stalled to assure consistency with the online monitoring, as explained below. Note that this stall does not lead to a noticeable performance loss, as the fine-tuning is only started after a computational hot spot has finished its execution, but not within the loop of a hot spot. Even the worst-case situation (i.e. an SI execution right after the fine-tuning was started) only leads to a loss of few cycles per hot spot.

Our \textit{Run-Time Manager} uses an online monitoring approach for fine-tuning the \textit{Forecast Values}. The concept of fine-tuning the \textit{Forecast Values} was illustrated in Fig. 8. In the following we concentrate on its realization and the interaction of FIs and SIs, as they are the main input to the \textit{Run-Time Manager} (besides statically provided information about SIs, data paths, etc). Whenever an SI is executed, a corresponding counter is increased until an upper limit is reached (to prevent an overflow). This counter reflects the execution count between \( F_1 \) and \( F_2 \) in Fig. 8b. When \( F_2 \) is reached, the difference between the counted executions and the forecasted executions from \( F_1 \) is computed as the \textit{error} of the \textit{Forecast Value} at \( F_1 \). This \textit{error} is then weighted and back propagated (i.e. accumulated) to \( F_1 \) (as shown in Fig. 8b) to improve the quality of the \textit{Forecast Value} at \( F_1 \), for the next iteration, i.e. to fine-tune the \textit{Forecast Value}. The details about different scaling factors to weight and back-propagate the \textit{error} are examined in [35].

In the case that a preceding fine-tuning operation is still processed when \( F_2 \) is reached, we need to wait until the preceding fine-tuning is finished. The reason is that the counter value would not be reset for the not-yet processed \textit{Forecast Values} (see Fig. 9), which then would lead to wrong counter values in the next loop iteration. The same is true if an SI executes while a fine-tuning operation is currently running (see “Synchronize Forecast and Special Instruction” in Fig. 9).

The asynchronous part of the \textit{Run-Time Manager} comprises the data path re-loading decisions (see Fig. 9). Although it is triggered by an FI, it can execute independently after it was started. In contrast to fine-tuning the \textit{Forecast Values} we may abort a previously started execution of this part, as the preceding FI is obsolete when a new one arrives. The \textit{Forecast Values} are used together with (at compile-time determined) information of the different SI implementations to select implementations for all requested SIs. As this decision has to be made at run time, the computational overhead is critical and we are therefore deploying a greedy heuristic. This heuristic iterates over all implementations of the forecasted SIs and determines a profit value for each one. The locally best implementation is selected and the same procedure is repeated for the remaining SIs. As soon as the first reconfiguration is started (after the first local SI selection), more time may be spent in determining further decisions, as it is then done in parallel to the reconfiguration. For the selected SI implementations, the data paths need to be loaded sequentially, as typically only one reconfiguration port is available. Determining a specific reconfiguration sequence may have a high impact on the application execution time. For instance, multiple SIs might be required for the upcoming hot spot and some of them might be executed significantly more often than others. In this...
case, it might be beneficial to reconfigure the data paths for the more often required SIs at first, depending on the performance differences of the corresponding implementations. Having the hardware implementation in mind, we decided to use a heuristic that locally selects those data paths that lead to the locally best performance improvement due to a better SI implementation. Finally, some currently available data paths may need to be replaced to offer new SI implementations. If nearly all data paths are going to be replaced, then the decision which one to keep has only a small impact. However, in situations where only relative few data paths are going to be replaced (e.g. due to insufficient reconfiguration time), it is beneficial to keep those data paths that will be required again soon. As the upcoming requirements are not known beforehand, we replace those data paths that lead to locally smallest performance degradation of the currently available SI implementations.

C. Hardware Prototype

Our envisioned RISPP targets an ASIC implementation for the core pipeline with a tightly coupled embedded FPGA (eFPGA, as investigated in [36]) for the reconfigurable region (so-called data path containers), see Fig. 4. For our current hardware prototype we implement both parts on an FPGA. The main purpose is to learn more about real-world constraints and the actual system behavior to increase the accuracy of our simula-

---

**Fig. 9: State-transition diagram of the Run-Time Manager**

- **Normal Instruction**
- **Special Instruction (SI) detected**
- **Forecast Instruction (FI) detected**
- **Execute the Special Instruction**
  - Examine available data paths
  - Send trap signal to CPU. Corresponding handler will execute the SI
- **Update the monitoring value**
  - Read current monitoring value for executed SI
  - Value reached upper limit
  - Value below upper limit
  - Increment the value and write it back
- **Fine-tuning running**
  - IF a fine-tuning process is currently running, THEN stall the core pipeline and the both beside processes
- **Synchronize Forecast and Special Instr.**
- **Fine-tune the Forecast Values**
  - Test, whether the preceding FI is still processed
  - Preceding FI finished
  - Updating not allowed & no more SIs to be processed
  - More SIs to be processed
  - Updating allowed
  - No more SIs to be processed
  - More data paths to be loaded
  - All data paths loaded
  - Schedule the next data path loading
  - All data paths loaded
  - Abort pending
  - Send abort-Signal to the previous FI processing
  - Preceding FI finished
  - Updating not allowed but more SIs to be processed
  - More SIs to be processed
  - Updating allowed
  - No more SIs to be processed
  - Compute a weighted error (diff. between Forecast Value and monitoring value)
  - Backpropagate this error to the preceding FI for the corresponding SI
- **Determine the data path reloadings**
  - Test, whether the preceding FI is still processed
  - Preceding FI finished
  - For all forecasted SIs
  - select an implementation
  - IF a fine-tuning process is currently running, THEN stall the core pipeline and the both beside processes
  - Updating not allowed but more SIs to be processed
  - More SIs to be processed
  - Updating allowed
  - No more SIs to be processed
  - Compute a weighted error (diff. between Forecast Value and monitoring value)
  - Backpropagate this error to the preceding FI for the corresponding SI
- **LEGEND:**
  - Start state
  - End state
  - Two sub state machines, executing in parallel
tion environment (e.g. accurate reconfiguration times, data path area requirements, etc.). Our current FPGA prototyping platform is based on a Xilinx board with additionally self-developed peripheral extensions. We partition the FPGA in one region that contains the static core pipeline and one region that contains the data path containers. Our core pipeline is based on a MIPS processor that was extended for our requirements and runs at 50 MHz. Note that for the later simulations we assume a frequency of 100 MHz to address the fact that the core pipeline will be implemented with ASIC technology.

In Fig. 10, we show the placed and routed floor plan of our design [34] using a Virtex-II xc2v3000 FPGA from Xilinx. In the right region, four partially reconfigurable data path containers are established and connected to the core pipeline via BusMacros (i.e. special Xilinx-provided soft IP-cores for communication across reconfiguration boundaries [37]). A reconfigurable data path container spans four CLB (Configurable Logic Block) columns and offers 1024 slices.

![FPGA Floor plan: a static region for core pipeline and a partially reconfigurable region for data path containers](image)

Table III shows the implementation results for the data paths shown in Fig. 10 (required to implement SATD and I/H/4x4; see Table I). The implementation was done using Xilinx PlanAhead with the Early Access Partial Reconfiguration (EAPR) tool flow. It is noticeable that the utilization of the data path containers is moderate (between 34% and 50%), but shrinking the size of the containers by only one CLB column leads to a situation where the synthesis tool fails to place & route the data paths into the area restrictions of the containers. However, this situation is FPGA specific and may be avoided when using an eFPGA with optimized routing resources (e.g. like investigated in [36]) within each data path container. The reconfiguration time corresponds to the size of the partial Bitstream and the reconfiguration bandwidth. The partial Bitstream is the amount of reconfiguration data that has to be transmitted and the maximum reconfiguration bandwidth for the SelectMap/ICAP interface of our Virtex-II FPGAs is 66 MB/s.

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>HT_4</th>
<th>SATD</th>
<th>Re-pack</th>
<th>QSub</th>
</tr>
</thead>
<tbody>
<tr>
<td># Blocks</td>
<td>20</td>
<td>40</td>
<td>400</td>
<td>52</td>
</tr>
<tr>
<td># LUTs</td>
<td>1034</td>
<td>808</td>
<td>512</td>
<td>700</td>
</tr>
<tr>
<td>Utilization</td>
<td>50.5%</td>
<td>39.5%</td>
<td>39.7%</td>
<td>34.2%</td>
</tr>
<tr>
<td>Gate Equiv.</td>
<td>8,599</td>
<td>6,645</td>
<td>6,731</td>
<td>5,599</td>
</tr>
<tr>
<td>Latency [ns]</td>
<td>6.48</td>
<td>8.34</td>
<td>3.54</td>
<td>3.50</td>
</tr>
<tr>
<td>Size [byte]</td>
<td>59,305</td>
<td>68,141</td>
<td>65,713</td>
<td>58,749</td>
</tr>
<tr>
<td>Reconf. Time [µs]</td>
<td>587.63</td>
<td>840.11</td>
<td>849.53</td>
<td>848.84</td>
</tr>
</tbody>
</table>

**Table III: Results for hardware implementation of individual data paths**

A. Comparing Data Path Utilization and Efficiency between ASIP and RISPP

Our proposed RISPP architecture is meant to be an ASIC (for the core pipeline) with an embedded FPGA (eFPGA) for the reconfigurable data paths, while an ASIP is completely implemented in ASIC technology. As there are fundamental differences between the ASIC and the FPGA technology, care has to be take when comparing both approaches. We will now enumerate the assumptions and similarities and highlight the differences of both architectures before discussing the fairness of our comparison.

**Assumptions & Similarities:**

i. We have used MIPS as the core pipeline for both ASIP and RISPP running at same frequency (100 MHz) along with same hardware resources (e.g. register file, read/write ports, memory accesses, periphery etc.).

ii. ASIP and RISPP used the same benchmark application and could access the same set of optimized data paths, SIs, and SI implementations. Each data path takes 1 cycle for execution. The execution time of the different SI implementations is determined accordingly.

iii. RISPP additionally uses an embedded FPGA that can be reconfigured to contain different data paths. For the reconfiguration time we have used the implementation results of our FPGA-based prototype (0.84 to 0.95 ms, see Table III). As the implemented data paths have almost similar sizes we consider them as our basic area unit.

**Dissimilarities:**

iv. ASIPs may achieve a higher clock frequency as the core pipeline and the data paths are completely implemented in ASIC technology. This is true for the core pipeline and the Run-Time Manager of RISPP as well, but the data paths for RISPP are implemented in a potentially slower reconfigurable fabric. However, the major bottleneck for the SI execution is not the frequency, but the limited memory bandwidth (e.g. the DCT_4x4 SI in Fig. 5 requires eight 32-bit inputs and writes eight 32-bit outputs). While at 100 MHz frequency a single cycle memory access is possible, at higher frequencies the memory access latency increases. Due to the streaming nature and the non-linear access pattern of e.g. the DCT computation, the benefit of data caches is limited. Therefore, the performance of the SIs, (and thus the major hot spots) does not linearly scale with increased frequency.

v. As the data paths for the ASIP are implemented in ASIC technology, they will require less area than those implemented in reconfigurable fabric. However, when using an embedded FPGA that is optimized for our data path containers, their implementation will be smaller compared to our current prototype. Our currently used FPGA is not designed for segmentation into (rather small) data paths containers and thus the routing re-
sources of the FPGA become a bottleneck. This is because many of the available routing resources cannot be used as they are crossing the boundaries between adjacent containers, instead of staying inside the container.

vi. As the ASIP data paths are implemented in ASIC technology, they typically cannot be reused for anything different from the initially considered application. Therefore, the ASIP has to offer data paths for all considered applications statically to accelerate them. For an increased amount of applications, the size of the ASIP may grow continuously. For instance, the presented H.264 video encoder is just one part of the H.324 video conferencing application (Fig. 1) that will be executed besides other applications like encryption or baseband processing. Additionally, the highest achievable frequency of the ASIP may be reduced due to the increased distance between the large amount of data paths and the core pipeline. RISPP instead can restrict to a smaller amount of data paths containers and then reconfigures them towards the currently required operations.

vii. The data paths for the ASIP in the later comparison are selected in an optimal way considering the performance. This does not only comprise exact knowledge of the application, but also exact knowledge of the input video sequence and thus the resulting SI execution frequencies (for (I)HT, 4x4, MC_Hz_4, and IPred_HDC it depends on the control flow; see the ‘then’ and ‘else’ part in Fig. 11). As in real-world scenarios the motion in the input data cannot be predetermined, the ASIP may not have the optimal data paths available, but it will have to encode the video with a suboptimal selection of data paths. RISPP instead dynamically adapts the SI implementation to cover the current type of motion in the input video sequence (using the Forecast Values, see Section III.B).

viii. The RISPP architecture demands a run-time system (see Section III.B) to determine the reconfiguration decisions. Even though this achieves dynamic adaptation it comes at the cost of a static area overhead. Implementing the synchronous part (see Section III.B) did not affect the area requirements significantly. However, implementing the asynchronous part results in additional area requirements that correspond to a few data path containers. Note, in a final ASIC with eFPGA implementation the Run-Time Manager belongs to the non-reconfigurable core pipeline and thus the area requirements will benefit from ASIC technology.

**Fairness of Comparison:**

We will now discuss the fairness of our cross-architecture comparison between ASIPs and RISPP. On one hand the ASIPs seems to be underestimated, as the data paths could be implemented in better technology (see iv, v) and they do not need a Run-Time Manager (see viii). On the other hand, RISPP seems to be underestimated as well, as it facilitates a significantly higher flexibility to support different applications (even in multitasking environments) without the predetermined knowledge which applications will be executed on which input data pattern (see vi, vii). Depending on the actual requirements and objective targets, some of these points might dominate the others. If the target application for instance uses rather few and small SIs, then an ASIP may be privileged. The comparison partner of the ASIP in this case should actually be a dedicated ASIC implementation and not RISPP. However, consider for instance a target system like a mobile device, where multiple applications have to be executed and the owner of the device can download and execute further applications. In these cases, the provided flexibility and dynamic adaptivity of our RISPP architecture will dominate the advantages of a tailor-made ASIP implementation, which can only cover a certain subset of the applications. This is because the ASIP does not scale with an increasing amount of target applications and it cannot address (at design time) unknown applications at all.

Taking into account that a processor which is fabricated according the RISPP architecture does not need to be re-fabricated when facing different applications, the RISPP selling volume can be expected to be higher than that of a tailor-made ASIP. This furthermore allows RISPP to use of a more sophisticated masking technology compared to a low- to middle-volume ASIP, as the non-recurring engineering cost will amortize for RISPP. Thus, the potential area- and frequency advantages of ASIC-implemented data paths for ASIP (see iv, v) might diminish (depending on the budget and expected selling volume).

Summarizing the whole comparison, for the core pipeline, RISPP may use the same fabrication technology as the ASIP and the main difference comes in the data paths. It will result in a different SI performance for ASIP and RISPP but this highly depends on which ASIP and RISPP technologies are benchmarked and thus it depends on the specific system requirements. Therefore, we aimed to achieve a neutral (and thus general) comparison by considering the cycle-count and the data path as performance and area measurement unit, respectively.

**B. Benchmark Application**

In order to obtain our benchmark application, the H.264 reference software [16] is passed through a series of application architectural optimizations as specified in [38] to improve the performance of the application on the targeted ASIP and RISPP platforms. The following gives an overview of these optimizations:

a) First, we adapted the reference software to contain only Baseline-Profile tools considering multimedia applications with Common Intermediate Format (CIF: 352x288) or Quarter CIF (QCIF: 176x144) resolutions running on mobile devices.

b) Afterwards, we improved the data structure of this application, e.g. by replacing multi-dimensional arrays with one-dimensional arrays.
c) We have then used an optimized low-complexity Motion Estimation (ME) process to reduce the ME processing load. This is our in-house developed ME with adaptive search patterns that ultimately results in at most 67 search candidates while giving comparable visual quality as of the exhaustive search (that requires 1089 search candidates).

d) We have decoupled the ME and Rate Distortion (RD) from the Encoding Engine. As ME can be processed independently on the whole frame, we take it off the Encoding Engine. We additionally take RD outside the Encoding Engine and perform an early decision to execute exactly one MB type and one prediction mode instead of both types with all 13+16 prediction modes [38].

Fig. 11 shows the application architecture of our H.264 video encoder after these optimizations. We then profiled this application and detected the hot spots. For each hot spot, we have designed and implemented several Special Instructions (composed of hardware accelerators as shown in Table I). This serves as our benchmark application, which is then used for both ASIP and RISPP. Our hardware accelerators and SIs are designed after an extensive and iterative optimization effort to make the experiments closer to the real-world scenarios. The same SI implementations were made available for the ASIP to make the comparison as fair as possible.

C. Results and Discussion

At first, we present the general trade-off for the different SI implementations in Fig. 12, as it was motivated in Fig. 5. The performance measurements of this trade-off are independent of an ASIP or RISPP implementation. The difference between both architectures is the fact that the ASIP has to choose one implementation statically at design time, whereas RISPP can make and modify its decisions at run time. Another feature of RISPP is to upgrade at run time from one SI implementation towards a faster one, which mainly corresponds to a movement on the Pareto-optimal front in Fig. 12. In general, even the non-Pareto-optimal implementations are relevant, depending on the data paths that already finished reconfiguration. For instance, the HT_4x4 implementation that only uses the Repack data path requires 184 cycles for one SI execution, while the implementation that only uses the HT_4 data path needs 104 cycles. Therefore, the implementation with the Repack data path is not Pareto-optimal, but still it is beneficial for the performance, whenever the Repack data path is already available (i.e. reconfigured), but the HT_4 data path is not available (i.e. not reconfigured yet).

![Fig. 12: SI performance for different implementations, facilitating dynamic trade-off](image)

In the motivation (Section II), we have introduced and analyzed the data path utilization (see equation 2) and efficiency of the data path usages (i.e. the speedup per available data path; see equation 1). Our analysis showed that the efficiency of ASIPs is rather moderate when only a few data paths are provided. However, to reach a good operation point with a high efficiency many data paths would need to be added. The small speedup when only a few accelerating data paths are provided is due to an inefficient utilization of the available hardware resources. We tackle this problem through our proposed extensible processor RISPP (Rotating Instruction Set Processing Platform) that uses the available hardware for data paths in a time-multiplexed way to implement modular SIs. When the processing of one hot spot is completed, its hardware is re-allocated to the SIs of the subsequent hot spots.

![Fig. 13: Application execution time and efficiency of resource usage for encoding 140 video frames on ASIP and RISPP](image)
The execution time of the application using a GPP without hardware accelerators is 7.4 billion cycles. It can be noticed from Fig. 13 that in case of five data path containers RISPP has the maximum efficiency of 3.6 (this is 3.0 times better than that of ASIP) as it can already execute all SIs in hardware (due to the time-multiplexed hardware usage). The execution time of the application in that case is 416.67 million cycles, i.e. 17.8 times better than GPP.

When using up to nine data path containers, the performance of RISPP is better than that of ASIP. We call this point “break-even point” for RISPP where both architectures have nearly the same efficiency and execution time because all SIs are executed completely in hardware. Beyond the break-even point, the reconfiguration delay dominates the performance gain of faster hardware implementations of SIs. ASIP starts winning beyond the break-even point, but after some further addition, a saturation point is reached to Amdahl’s law. RISPP instead achieves already a good performance with a high efficiency for a rather small number of available data paths. Table IV summarizes the important attributes of the comparison of RISPP and ASIP.

Table IV: Summary of comparison of RISPP and ASIP

<table>
<thead>
<tr>
<th></th>
<th>ASIP</th>
<th>RISPP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exec. Time [MCycles]</td>
<td>min</td>
<td>avg</td>
</tr>
<tr>
<td>Speedup vs. GPP</td>
<td>2.4</td>
<td>16.8</td>
</tr>
<tr>
<td>Efficiency</td>
<td>1.0</td>
<td>1.3</td>
</tr>
</tbody>
</table>

Fig. 14 demonstrates the resource utilization (see equation (2)) of RISPP and ASIP. The figure shows that the resource utilization of RISPP is much better than that of ASIP for a small number of data path containers due to the time-multiplexed utilization of the hardware resources. We will now analyze three significant points (see the labels in Fig. 14) with the help of a corresponding run-time analysis (Fig. 15) in detail. Label A in Fig. 14 represents the best case of resource utilization (57.4%) corresponding to four data path containers for RISPP. Fig. 15a shows the detailed data path utilization for this case. In the first functional component (i.e. Motion Estimation (ME); see Fig. 11) all four data path containers are used. After ME has completed execution, RISPP reconfigures the data paths for the next component (i.e. Encoding Engine), as can be seen at the temporary drop of the resource utilization.

DCT_4, Point Filter, and Clip3 get loaded in place of HT_4, SATD_16, and QSub to support the DCT_4x4 and MC_Hz_4 SIs in hardware (timeframes 23 & 24). In timeframe 23 RISPP uses more than four data paths, as in the beginning of this timeframe the data paths for the ME are still available and used, where at the end of this timeframe the data paths for the Encoding Engine are already loaded and used. However, at no time more than four data paths can be used together. In timeframe 29 & 30 the data paths are reconfigured towards the third functional component (i.e. Loop Filter), before the execution of the three components restarts for the next input frame (not shown for clarity). It is noticeable that the ASIP instead spends all four data paths for the ME (see Fig. 15a left side), as this enables the best performance for the overall encoding. In timeframe 34 (within the Encoding Engine), the ASIP reuses the HT_4 and Repack data paths to encode some I Macroblocks (I-MBs). RISPP instead reconfigures the data paths to support the more frequent P-MBs and accepts using the software method to encode the I-MBs. By doing so, RISPP achieves an encoding time that is significantly faster than that of the ASIP. RISPP executes ME slightly slower than the ASIP due to the initial reconfiguration overhead, but the Encoding Engine and Loop Filter overcome this shortfall.

Label B in Fig. 14 portrays the worst-case resource utilization (11%) corresponding to five data paths for ASIP. This is because the ASIP additionally offers the SATD data path (see Fig. 15b) that can only be used for the SATD_4x4 SI (see Table I). This leads to the best performance (6 instead of 23 timeframes for ME), but the Encoding Engine and Loop Filter do not benefit at all and now dominate the execution time. At Label C in Fig. 14 the data paths utilization between RISPP and ASIP is similar. Now also for the ASIP all three functional components are covered with hardware accelerated SIs (although not with the fastest implementations).

Fig. 15c shows that the RISPP architecture is already affected by the reconfiguration time. The first four timeframes are spent in reconfiguring all selected data paths for the ME hot spot, which does not only have a negative impact to the utilization, but also to the performance. The same is true for the Encoding Engine and partially for the Loop Filter. Therefore, only in timeframe 10 all provided data paths are actually used. Finally, the better SI implementations that RISPP selected no longer overcome the reconfiguration overhead. An improved reconfiguration time would attenuate this effect and further improve the performance and data path utilization of our RISPP architecture.

---

**Table IV: Summary of comparison of RISPP and ASIP**

<table>
<thead>
<tr>
<th></th>
<th>ASIP</th>
<th>RISPP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exec. Time [MCycles]</td>
<td>min</td>
<td>avg</td>
</tr>
<tr>
<td>Speedup vs. GPP</td>
<td>2.4</td>
<td>16.8</td>
</tr>
<tr>
<td>Efficiency</td>
<td>1.0</td>
<td>1.3</td>
</tr>
</tbody>
</table>
**CONCLUSION**

We have presented the adaptive extensible processor **RISPP**, which provides a high utilization and efficiency of resources through run-time adaptation of the instruction set. It goes beyond the capabilities of state-of-the-art extensible processor paradigms that fail to provide efficient solutions for highly complex applications. Our approach uses **modular Special Instructions** in a time-multiplexed manner, i.e. the **Special Instructions** are decomposed into elementary data paths, which can be reconfigured as required. It enables us to achieve a 3.0 times better efficiency (avg. 1.4x) of resource usage than state-of-the-art **ASIPs** resulting in a 3.1x (avg. 1.4x) improved performance. Compared to a **GPP**, we achieve a 25.7x (avg. 17.6x) performance improvement. Additionally, we increase the flexibility of our **RISPP** architecture, which is no longer tailor-made towards a specific application or application domain, but generic for different kinds of application domains.

**REFERENCES**


[31] MiBench (http://www.eecs.umn.edu/mibenche/).


Lars Bauer received his M.Sc. degree in Information Technology from the University of Karlsruhe (TH), Germany in November 2004. He is pursuing his Ph.D. at the Chair for Embedded Systems (CES), University of Karlsruhe (TH) under the advice of Prof. Dr. Jörg Henkel. His main research interests are extensible and reconﬁgurable computing systems with a focus on dynamically varying run-time situations and concepts that allow systems to adapt to changing requirements. Since 2005, Lars Bauer is an IEEE Student Member.

Muhammad Shafique received his M.Sc. in Information Technology from Pakistan Institute of Engineering and Applied Sciences in 2003. He had been awarded with 6 Gold Medals and Best Master Thesis award. He then joined Streaming Networks as Senior Embedded Systems Engineer where he was involved in the development and optimization of H.264 Encoder on VLIW processors. Since October 2006 he is pursuing his Ph.D. at the Chair for Embedded Systems (CES), University of Karlsruhe (TH) under the supervision of Prof. Dr. Jörg Henkel. His main research interests are extensible and reconfigurable processors for low-power mobile multimedia with a focus on dynamically varying run-time situations. He holds one US patent. Since 2007, Muhammad Shafique is an IEEE Student Member.

Dr. Henkel received his Master and Ph.D ("Summa cum laude") degrees both from the Technical University of Braunschweig, Germany. He then joined the Computer & Communication Research Laboratories CCR (now NEC Laboratories America) in Princeton, NJ, where he led various projects in the areas of low power system level design and advanced embedded architectures. In between, he had an appointment as a visiting professor at the University of Notre Dame, IN. Dr. Henkel is currently a Full Professor at Karlsruhe University (TH), Germany, where he is directing the Chair for Embedded Systems CES.

Dr. Henkel is actively involved in many international conferences and journals etc. For example, he is the Editor-in-Chief of the ACM Transactions on Embedded Computing Systems (ACM TECS). He is also an Associate Editor of the IEEE Transaction on VLSI Systems (TVLSI). In addition, he is the Chairman of the IEEE Computer Society, Germany Section. He serves on various steering committees like Codes+ISSS, the JOLPE Journal on low power electronics etc. As a program committee member he is serving or has served for all major conferences in electronic design automation and embedded system design including DAC, ICCAD, DATE and others. He has given full-day tutorials at DAC, ICCAD, DATE and others in the area of embedded system design. He was the chair of the PhD Forum at the DATE Conference in 2006 and 2007. In 2001 he served as a Program Chair for the IEEE/ACM Codes Hardware/Software Co-design Symposium and was a General of the same convention in 2002. Furthermore, he was a Program Chair for the 2002 IEEE Workshop on Rapid System Prototyping. In 2006 he served as a Program Chair for the IEEE/ACM Symposium on Low Power Electronics and Design (ISLPED). Currently (2008), he is serving as a program Chair for the IEEE SIPS workshop on Signal Processing. He has guest-edited special issues on hardware/software co-design in the IEEE Computer Magazine and on rapid system prototyping. Since 2008 Dr. Henkel is an Executive Committee Member of ICCAD. Recently (in June 2007) his edited book “Designing Embedded Processors – A Low Power Perspective” was published by Springer. He holds eight US patents.