Automatic Custom Instruction Identification in Memory Streaming Algorithms

Martin Haaß, Lars Bauer and Jörg Henkel
Karlsruhe Institute of Technology (KIT)
{martin.haass, lars.bauer, henkel}@kit.edu

ABSTRACT
Application-specific instruction set processors (ASIPs) extend the instruction set of a general purpose processor by dedicated custom instructions (CIs). In the last decade, reconfigurable processors advanced this concept towards runtime reconfiguration to increase the efficiency and adaptivity. Compiler support for automatic identification and implementation of ASIP CIs exists commercially and on research platforms, but these compilers do not support CIs with memory accesses, as ASIP CIs typically work on register file data. While being acceptable for ASIPs, this imposes a limitation for reconfigurable processors as they achieve their performance by exploiting data-level parallelism. Consequently, we propose a novel approach to CI identification for runtime reconfigurable processors with support for memory operations in contrast to previous works that explicitly exclude them. Our algorithm extracts memory access patterns which allows us to abstract from single memory operations and merge accesses to optimally utilize the available memory bandwidth. We implemented our algorithm in a state-of-the-art compiler framework.

The largest CI identified in our benchmarks consists of 2071 nodes (average 999 nodes), and a single generated CI can cover a whole computational kernel (up to 99%).

Categories and Subject Descriptors
C.0 [GENERAL]: Instruction set design

General Terms
Design, Algorithms, Performance

Keywords
Reconfigurable architecture, custom instruction generation, load/store merging, streaming memory access

1. INTRODUCTION
As more and more functionality moves from desktop systems to embedded systems the demand for low-power, low-cost embedded systems grows. These low-cost systems are supposed to process demanding tasks like video encoding/decoding, cryptography and more. Many of those tasks need hardware accelerators to achieve the necessary performance which was traditionally done by manually designing application specific integrated circuits (ASIC). Such a time consuming process does not fit with low-cost systems and their short time-to-development cycles. This lead to application-specific instruction set processors (ASIPs) that consist of a general purpose processor core (GPP) which is extended by one or more application specific custom instructions (CIs). These CIs are executed in custom functional units (CFUs, [1]) which can be implemented using ASIC technology [2, 3, 4] as well as on reconfigurable platforms [5, 6, 7].

A CI implements the specific functionality of an application or application domain and allows the GPP to offload tasks for which it is not suited to the CI hardware implementation, that allows for considerable performance improvements. It consists of a set of basic operations which are mapped onto the available structures within the CFU using off-the-shelf hardware synthesis tools. By bundling many operations into one CI and replacing those operations in the original code by a single CI instruction, the code size can be reduced and the application runtime benefits due to the parallelism within the CI.

CIs are generated at compile time in a two-step process. The first step enumerates legal candidates (typically a large set) and the second step selects the final CIs out of the set of candidates while paying attention that selected candidates do not overlap, because this would lead to CIs that cannot be scheduled on the hardware. Instead, our proposed approach addresses both steps in a single CI identification process. A typical compilation flow with source code as input and data flow graphs (DFG) as intermediate representation (IR) is shown in Figure 1 and the two steps that can be merged due to our approach are briefly explained in the following.

A straightforward approach for the CI enumeration subproblem enumerates all nodes in a DFG and combines them into CIs until some constraint (e.g. area) is violated. This is obviously difficult as the amount of possible subgraphs grows exponentially with the number of operations in the DFG [8]. Other approaches that target the CI enumeration subproblem are either optimal but imply more than polynomial execution time for the worst case [9, 10, 11] or bound the execution time and sacrifice optimality instead [8, 12, 13].
Common to both approaches for the CI enumeration sub-problem is the large number of generated candidates which typically ranges in the thousands for small computational kernels like the Rijndael cryptographic function [11]. From this vast amount of candidates, the CI candidate selection sub-problem has to determine a subset that performs best with regard to some criteria, like energy consumption or application execution time, and it has to map this subset to all occurrences within the application while ensuring that selected candidates do not overlap. Avoiding overlapping candidates is a graph covering problem, i.e., an NP-complete processing step [14]. Different approaches exist to solve the CI selection sub-problem. They typically use greedy and subset-sum approaches in combination with other techniques like pruning [14, 15, 16].

In this paper, we propose a novel CI enumeration approach that identifies CIs in the range of only tens to hundreds for one application compared to the number of thousands of candidates that are enumerated and need to be processed in literature [8, 11, 17]. Due to the way candidates are generated in our algorithm the resulting CIs inherently do not overlap. Those two properties together (i.e., small number of non-overlapping candidates) allow us to eliminate the separate candidate selection step. Instead, we can directly select all candidates identified by our enumeration algorithm.

Most of the previous work formulate a set of forbidden nodes which are not allowed to occur as part of a CI candidate because the assumed hardware platform does not support them. There are different ways in coping with the occurrence of such forbidden nodes, e.g., Atasu et al. [18] cluster them and identify ancestors which need to be removed from the set of possible nodes, whereas Li et al. divide the search space before the search continues and thus exclude the forbidden node [19]. Common to all these approaches is that load and store operations are explicitly excluded from being used in CI candidates. This restricts how CIs can access data and means that data has to be fetched and temporarily stored before it can be used within a CI. As a result, the authors aim for wide register file access and benchmarks are conducted for setups with up to 7/3 concurrent read/write ports [17]. As Yu et al. explain, these constraints are due to the data being stored in the GPP register file or in explicit transfer registers because the architectures do not permit direct memory access of the CI [10].

In this paper, we propose a novel algorithm for identifying CIs which can access on-chip memory without intervention of the GPP. The identified custom instructions are able to access a noticeably bigger amount of data than available in the GPP register file. By including memory operations like load/store into the generated CIs we can reduce the list of forbidden nodes and cover larger subgraphs of the application which are replaced by a CI, resulting in smaller code and enhanced application performance for memory-intensive inner loops by executing the identified CI on a CFU.

The rest of this paper is organized as follows: The next section introduces related work. Section 3 gives an overview of typical architectural features that are necessary to allow CIs to access main memory. The problem formulation is given in Section 4 and Section 5 details our algorithm. Section 6 presents the results and Section 7 concludes the paper.

2. RELATED WORK

Previous works can be divided into enumeration algorithms that explicitly forbid memory accesses and into approaches that allow enumerated CIs to access memory. Different heuristic approaches exist that do not allow memory access like [1, 8, 11, 12, 13, 17, 18]. There are also exact methods which do exhaustive enumeration of all possible subgraphs [9, 15, 19]. All these approaches suffer the same limitation that the amount of generated candidates easily reaches one thousand per application and no consideration is given on how the CI selection process could be accomplished. Furthermore, by explicitly excluding memory accesses those algorithms limit the form of a candidate purely to arithmetic operations and are dependent on the constrained number of register file ports in their respective architectures. By excluding memory accesses these previous works have to achieve their speedup by combining the most time-consuming arithmetic into CIs whereas our new approach also exploits the reduction in load and store operations by efficiently combining successively memory operations.

The second class allows for memory operations and each work deals in a unique way with the additional challenge imposed by having load and store operations in a custom functional unit (CFU). Biswas et al. [20] were the first to support memory accesses in CIs by including CFU-internal memory to reduce access to register file, scratchpad and background storage. Here, all input data has to be transferred to the internal memory before the CI can start and copied out again after it is finished. In the duration between data transfer and CI execution the data is locally stored within the CFU and thus it is called a state-holding CI. Biswas et al. notice that it clearly reduces the register file pressure but they do not measure the area overhead introduced by having memory components in every CI, which severely constrains the size of internal storage. To fill this internal storage, code modifications are inserted which can only transfer whole chunks of data in a separate DMA phase potentially transferring unneeded data elements or, in the case of scalars, not reducing memory accesses at all.

Lee et al. [21] start from the directed acyclic graph (DAG) that represents the program and they augment it manually with dependency edges that serialize the memory operations. This shifts the bottleneck from register ports to sequential execution of load or store operations, as the authors note in the paper. Their solution is to re-parallelize memory operations by allowing directly consecutive memory operations to be performed in parallel. Without a strict order on the accessed addresses, memory aliases (i.e., addresses which are used by more than one pointer) can occur and are subject
to problems like e.g. read-after-write (RAW) where a read access of the first pointer is erroneously postponed until the value has been overwritten by a write access of the second pointer. Lee et al. therefore propose a data cache with an explicit alias handling logic that resolves these concurrent access situations. Their underlying CI enumeration uses the ‘single cut’ algorithm from [22] and thus suffers from the same computational complexity.

3. SYSTEM OVERVIEW

All known previous works in identifying custom instruction (CIs) for ASPs exclude memory operations (memops) in one way or the other and thus focus on architectures which do not support them. In this paper we provide compiler support for reconfigurable architectures that allow CFUs to directly access the main memory and we provide a short overview of such architectures. Reconfigurable architectures combine a GPP with a fine-grained runtime reconfigurable fabric (i.e. an embedded FPGA) that is closely coupled to the GPP, implements the CFUs, and has direct access to the main memory. Memory access is facilitated through two additional components: load/store units (LSUs) transfer data between main memory and CFUs, and address generation units (AGUs) calculate the addresses used to access the main memory.

Examples for such architectures are e.g. KAHRISMA [5], where memory can be accessed by coarse-grained encapsulated datapath elements (EDPEs) and where bit-/byte-level operations that work on this data are reconfigured into one or multiple fine-grained EDPEs. Other examples are RISPP [6] and the i-Core [7], which are runtime adaptive processors that grant memory access to their fine-grained reconfigurable CFUs.

While CFUs implement the computational data paths of CIs on the reconfigurable fabric, LSUs are implemented using the same non-reconfigurable technology as the GPP. The reason is that many CIs demand memory access making it more efficient to implement dedicated non-reconfigurable LSUs rather than to expect that every CI has its own access to the memory hierarchy which would consume area in its CFU. An overview of such architectures is given in [23]. With its programmable AGUs and LSUs, the i-Core is well suited for the kind of hw accelerators targeted in this work.

Input/Output data for CIs are often stored in on-chip scratchpad memory (SPM) which has reduced latency and larger bandwidth compared to the main memory as it is closer situated to the GPP and CFUs. SPMs typically have a direct connection to the SPM and can access \( W_{LSU} \) bits per memory operation. LSU ports with a bit width of \( W_{LSU} = 128 \) (or even more) furthermore enhance the memory bandwidth compared to the GPP’s load/store instructions.

AGUs can typically be configured by a base address and the parameters stride, span and skip. These 4 values altogether describe a particular memory stream \( m_i \in M \), i.e. a sequence of addresses to be accessed. The parameters allow supporting typical memory access patterns, as presented in [24]. The first address provided by the AGU is always determined by the base address and the next addresses are calculated by continuously adding either span or skip to the previously calculated address. Stride and Skip are the first-level and second-level offset between two subsequent addresses, respectively, and they can be positive or negative. Span is a positive value (or it is zero if skip is not needed) that describes how many addresses are accessed before adding skip rather than adding stride.

(a) 2D sub-array

(b) Arrangement of data elements in linear address space

Figure 2: Memory access pattern generated using the stride, span, skip parameters of the AGU

The AGU parameters allow to access a 2D sub-array of a 2D array as it is often required in video and image processing algorithms. Figure 2(a) shows an example 2D sub-array and Figure 2(b) shows how the data elements are arranged in the linear address space to illustrate how the AGU parameters are used to access them. Here, span defines the width of the sub-array and skip is used to jump to the next row in the sub-array, skipping data of the array that is not part of the sub-array. The 2D sub-array can also be accessed one column after the other (as e.g. required for matrix multiplication). For this, stride is set to the width of the 2D array, i.e. the next address points to the following row of the same column. At the bottom of the sub-array, after ‘span’ elements were accessed, a negative skip is added to go back to the second element of the first row of the 2D sub-array. The AGU parameters can be used to access linear arrays in a byte-wise manner (stride=1, span=0, skip=0). To access a linear array using an \( W_{LSU} = 128 \) bit port, the stride has to be set to 8 to calculate the byte address of the next access (stride=8, span=0, skip=0). The stride parameter can also be used to extract only one element out of a compound structure like, e.g. a pixel consisting of three color values red, green and blue (stride=3, span=0, skip=0).

The challenge that is addressed in this paper is to automatically enumerate suitable CIs that exploit the features and bandwidth offered by the statically defined LSU and AGU, i.e. to detect memory access patterns in the application, to configure the AGUs correspondingly, and to extract the affected data flow as CIs.

4. PROBLEM FORMULATION

To automatically identify CIs that directly operate on the main memory, we extend a state-of-the-art compiler framework that is based on the static single assignment (SSA) form [25].

Our proposed algorithms operate on the compiler IR, which is represented as one combined control and data flow graph (CDFG) per function in the source code. The CDFG consists of basic blocks (BBs) in which the total order of evaluation can be relaxed to a partial order [25] allowing for a high degree of instruction level parallelism which will be utilized for generating CIs. Operations that alias a memory
location (like two pointers to the same address) are still kept in correct order by extra memory dependency edges and, if unavoidable, synchronization points. The control flow determines the BB execution sequence and within a BB the data flow between operations is given. We apply loop unrolling and IF-conversion (i.e., replacing basic if-then-else control flows by data flow with a multiplexer node that selects the output) on the IR to increase the BB size and thus their inherent parallelism (see ‘Code Optimizations and Profiling’ in Figure 1). Profiling is used to identify and ensure that only computationally relevant BBs are processed, and it is ensured that no forbidden nodes are part of the BB. Where possible, function calls are Inline by the compiler previous to our analysis. BBs that contain a function call which could be possible, function calls are inlined by the compiler previous to our analysis. BBs that contain a function call which could not be Inline (e.g., library calls or system calls) cannot be transformed into a CI, as it is not possible to switch execution control to the CPU before the active CI is finished. Thus remaining call nodes are treated as forbidden nodes.

A BB is a directed acyclic graph \( G = (V, E) \) consisting of a set of nodes \( V \) representing basic operations that are connected by a set of directed edges \( E \subseteq V \times V \), where \( v_i = (v_a, v_b) \) represents a data flow dependency from node \( v_a \) to \( v_b \). We determine the external input/output data flow to/from the basic block \( G \) by the nodes \( V_{ext} \) and the edges \( E_{ext} \subseteq V_{ext} \times V \times V_{ext} \). For any subgraph \( (V', E') = G' \subseteq G \) we define \( IN_R(G') \) and \( OUT_R(G') \) as the number of inputs from \( G \) to \( G' \) and outputs from \( G' \) to \( G \), respectively, as shown in Eq. (1).

\[
IN_R(G') = \left\{ v \in V : \exists v' \in V' \exists (v, v') \in E \cup E_{ext} \right\} \\
OUT_R(G') = \left\{ v' \in V' : \exists v \in V \cup V_{ext} \exists (v', v) \in E \cup E_{ext} \right\}
\]

We further define the predecessors \( PRED(n) \) and successors \( SUCC(n) \) of a node \( n \in V \) as shown in Eq. (2). An example is shown in Figure 3, where \( PRED(n2) = \{n1\} \) and \( SUCC(n1) = \{n2, n3\} \). Note that \( n4 \notin \{n1, n2, n3\} \), because \( n4 \notin V \), i.e. \( n4 \) is an input to the highlighted subgraph, but it is not part of it (\( n4 \in V_{ext} \)).

\[
PRED(n) = \left\{ v \in V : (v, n) \in E \right\} \\
SUCC(n) = \left\{ v \in V : (n, v) \in E \right\}
\]

3. the number of different memory streams (see Section 3) in \( G' \) must not exceed the number of available AGUs, i.e. \( |M| \leq N_{AGU} \).

The convexity constraint (as shown in Figure 3) of the first condition ensures that the generated CI can be executed atomically. The second constraint ensures that its input/output data (except memory accesses) can be handled by the register file and the third constraint ensures that sufficient AGUs are available to handle all memory streams. Note that the register file read/write ports are typically limited as they are costly (e.g. 2/1 or 4/1 read/write ports). Intermediate values that are only needed inside the CI are therefore not written to the register file, but they are stored in individual registers which are part of the CI and that are tailored to the values which need to be stored.

All memory access patterns that can be generated by the AGU with its three parameters \( stride, span \) and \( skip \) are described by Eq. (3), where \( Address(j) \) corresponds to the \( j \)th address of the memory stream \( m_i \in M \) that is requested from the AGU which handles \( m_i \). The idea behind Eq. (3) is that basically at each access, the value of stride is added to the previously calculated address. However, after every span access, \( span \) steps, the value of span is added rather than the value of stride, which is expressed by the correction term in the last line. Eq. (4) shows the same equation (for the span \( \neq 0 \) case) solved for stride and skip.

\[
\begin{align*}
\text{span} &= 0 : \\
\text{Address}(j) &= \text{baseaddress} + j \ast \text{stride} \\
\text{span} &= 0 : \\
\text{Address}(j) &= \text{baseaddress} + j \ast \text{stride} + \lfloor j / \text{span} \rfloor \ast (\text{skip} - \text{stride}) \\
\text{Address}(j) &= \text{baseaddress} + \text{stride} \ast (j - \lfloor j / \text{span} \rfloor) + \text{skip} \ast \lfloor j / \text{span} \rfloor
\end{align*}
\]

5. CUSTOM INSTR. IDENTIFICATION

The algorithm proposed in this paper is based on enumerating and clustering of memory operations (memops) to deriving memory streams from them. Basic blocks without memops are not considered as the achievable speedup of clustering memops and their associated arithmetic operations easily outweighs accelerating arithmetic operations on the limited input/output data from the register file. An overview of our approach is shown in Algorithm 1 and described in the following.

\[
\begin{align*}
V' &= \text{loads} \cup \text{stores} \cup \left\{ v \in V : \forall i \in \text{loads}, \exists v_i \in \text{stores}, \\
& \exists x_1, ..., x_i \in V, \exists y_1, ..., y_i \in V, \\
& (l, x_1) \in E, (x_1, x_2) \in E, ..., (x_i, v) \in E, \\
& (v, y_1) \in E, (y_1, y_2) \in E, ..., (y_j, s) \in E \right\}
\end{align*}
\]

After extracting the load and store operations in lines 5 and 6 of Algorithm 1, the threshold \( N_{\text{load}} \) in line 7 is used to filter out basic blocks that would not exploit the memory bandwidth advantage of the LSU and thus lead to suboptimal CIs. The value for \( N_{\text{load}} \) is determined in Section 6.1. In lines 9 to 16 each memop is inserted into a...
Algorithm 1: Main Loop
1 foreach $IRG$ // IR Graph
2 do
3 | foreach $(V, E) \in BB$ $\in IRG$ // Basic Block
4 do
5 | $loads = \{v \in V : v.type = load\};$
6 | $stores = \{v \in V : v.type = store\};$
7 | if $|loads| < N_{def} \land |stores| < N_{def}$ then
8 | continue;
9 | foreach $v \in loads$ do
10 | normalizeAddressCalculation($v$); // see Algorithm 2
11 | extractAddress($v$); // this initializes $v.\text{baseAddress}$ and $v.\text{offset}$
12 | bucketsAGUcomp[$v.\text{baseAddress}$].insertByOffset($v$);
13 | foreach $v \in stores$ do
14 | normalizeAddressCalculation($v$);
15 | extractAddress($v$);
16 | bucketsAGUcomp[$v.\text{baseAddress}$].insertByOffset($v$);
17 | bucketsAGUcompatible = true;
18 | foreach $bucket \in \text{bucketsAGU} \cup \text{bucketsAGU}$ do
19 | mergeMemops($bucket$); // see Algorithm 3
20 | determineAGUParameters($bucket$);
21 | if $\text{bucketsAGUcompatible} = \text{false}$ then
22 | continue;
23 | // Extract CI subgraph, see Eq. (5)
24 | if $\text{stores} = \emptyset$ then
25 | stores = $\{\text{return}\}$;
26 | $V' = \text{extractClSubgraph}(loads, stores)$;
27 | $E' = \{e = (n, m) : n, m \in V', (n, m) \in E\}$;
28 | $G' = (V', E')$;
29 | if $\text{IN}_{\text{G}}(G') > N_{\text{inR}} \lor \text{OUT}_{\text{G}}(G') > N_{\text{outR}}$ then
30 | continue;
31 | $BB = \text{insertCustomInstruction}(BB, G')$;

bucket data structure (similar to a hash map) based on its type (load or store) and based on the base address that was used for its address calculation. Within a bucket, entries are sorted by the offset to the base address, which allows efficient clustering during the ‘mergeMemops’ phase (see line 19, explained in Section 5.2) before extracting the AGU parameters (i.e. stride, span and skip) in line 20). If the AGU parameters could not be determined for any of the buckets, this basic block is skipped in line 22.

Altogether, $N_{\text{AGU}} + 1$ buckets are used. The first $N_{\text{AGU}}$ buckets are used to capture the first $N_{\text{AGU}}$ memory streams (i.e. differing in their base addresses) encountered in the basic block. The additional bucket is used to capture all further memops for which the hardware architecture does not provide a dedicated AGU (i.e. the parameter $N_{\text{AGU}}$ is architecture specific). If a basic block contains more than $N_{\text{AGU}}$ memory streams, then the selection which of them should be calculated by an AGU would be relevant. We leave this extension for future work as we did not observe a basic block with more than 4 memory streams.

Finally, lines 23 to 27 create the CI subgraph. The basic idea is to include all load and store nodes and all other nodes that are on a path between a load and a store node (see Eq. (5)). In case there are no store nodes, the dedicated return node that denotes the end of the basic block is used instead. The construction of the CI as all nodes between two sets of nodes assure that it is convex by design. The address calculations for the load and store nodes are not part of the CI (the AGUs do that). They are removed from the basic block and are replaced by code for AGU initialization.

5.1 Address Normalization

To analyze address calculations we employ a recursive algorithm that traverses the address calculation of a given memop in a reversed depth-first manner (i.e. starting from the final address calculation node traversing upwards to the base address) and applies transformations as it descends. Figure 4 shows load operations and their respective address calculations as an example. Except basic cases without address calculation, the final node that delivers the address to the memop is always an addition. In the following, we distinguish the two inputs of this final addition as its left and right input.

Algorithm 2: Address Normalization
1 normalizeAddressCalculation Input: memop $\in V$
2 begin
3 | if $\text{SUCC}(\text{memop}) = i$ then
4 | duplicateSubgraph(memop);
5 | forall the $\text{pred} \in \text{PRED}(\text{memop})$ do
6 | normalizeAddressCalculation(pred);
7 | applyTransformations(memop);

The Goal is to create a canonical form where variables or symbols whose value cannot be determined at compile time are on the left input, whereas further operations or constants are on the right input. During this normalization, compiler optimizations like strength reduction and common sub expression elimination (CSE) have to be reverted. CSE can lead to subgraphs like the one shown in Figure 4 where the result of the upper Add node is consumed by more than one node, limiting the possibility to further normalize this subgraph. To achieve our canonical form for the address calculation of Load1, it is necessary to exchange the left const with the variable, which would change the address value that is calculated for Load2. In those situations the subgraph needs to be duplicated (see line 4 in Algorithm 2) to ensure that it is only used for one address calculation before applying further transformations for normalization.

Compared to disabling CSE, it is advantageous to revert it in the special cases for memops because we still profit from extensive constant propagation and static evaluation that are performed in earlier compilation phases. The resulting normalized address subgraphs for all memops of a memory stream are then used to extract the parameters for the AGU. As indicated in Figure 4 and explained in Section 5.3, these parameters are not necessarily compile-time known constants, but might depend on variables whose values are only known at runtime.

5.2 Memory Operation Merging

In this phase, multiple memory accesses (typically of bit
width 8, 16, or 32) are merged into one LSU access of bit width \( W_{LSU} \) (a so-called multi-word memop, MWM) to exploit the available bandwidth. Merging is done independently for each bucket in a greedy fashion exploiting the fact that the elements in the buckets are already sorted by their offset to the base address, ascending from smallest to largest. Algorithm 3 creates a new bucket (line 2) that replaces the old bucket at the end (line 16). The algorithm greedily absorbs one memop after the other into an MWM as long as they do not exceed the LSU bitwidth \( W_{LSU} \). Only if a memop exceeds the upper limit offset that the current MWM can access (line 7), it is inserted into the new bucket (line 8) and a new MWM is created (line 11).

**Algorithm 3: Merging of memory operations**

```
mergeMemops: Input: bucket[memops]
begin
    newBucket = new Bucket;
    MWM = NULL;
    foreach memop \in bucket do // sorted by offset
        upperLimit = memop.offset + memop.width - 1;
        // width = #Bytes accessed
        if MWM \neq NULL and
            upperLimit > MWM.upperLimit
        then
            newBucket.insertByOffset(MWM);
            MWM = NULL;
        if MWM = NULL
        then
            MWM = new MultiWordMemop;
            MWM.offset = memop.offset;
            MWM.upperLimit = MWM.offset + (W_{LSU}/8) - 1;
        if MWM \neq NULL
        then
            newBucket.insertByOffset(MWM);
        bucket = newBucket; // return via input param
```

Figure 5 shows this behavior for a \( W_{LSU} \) of 128 bit and four 32 bit load operations. The 32 bit data element between the second load (+32) and the third load (+96) is not used. The algorithms skips this hole and continues with the subsequent data element, i.e. it merges the first three memops into one MWM. The MWM actually includes the unused 32 bit data element, but it is not selected (right half of Figure 5). The last load in Figure 5 (b) has the offset +160 and thus does not fit into the first MWM anymore so an additional MWM is created.

**Figure 5: Merging of loads. (a) illustrates the array in memory, (b) shows how holes can be skipped and surplus loads are put into a new bucket**

### 5.3 Memory Stream Generation

To generate memory streams from a list of MWMs means to cover all memops with a common set of memory stream parameters. Memory stream parameters to initialize an AGU are baseaddress, stride, span, and skip, as shown in Eq. (3). In our compiler, the normalized address calculations have a form as shown in Eq. (6), where \( a \) and \( b \) are constants and baseaddress and \( x \) are variable values that are calculated at runtime. The normalized form of accesses sequences needs to be transformed into AGU parameters.

\[
\text{Address} = \text{baseaddress} + ax + b
\]

Table 1 shows the memory accesses of a real-world example that was created by compiling the sum of absolute differences (SAD) kernel of a 16×16 pixel subimage of a larger image. The first column shows the memop number, i.e. the position of the memop in the sorted bucket (sorted according to the memops offset, like in Algorithm 1). The second column shows the normalized address calculation that was generated by the compiler according to Eq. (6). The third column shows the MWM number, i.e. it denotes which memops are merged into the same MWM for an LSU with bitwidth \( W_{LSU} \). Each MWM is represented by its first memory access, i.e. MWMs #1, #2, and #3 are represented by memops 0, 4, and 8, respectively.

Columns 4 and 5 of Table 1 show the values \( \Delta a \) and \( \Delta b \) that are used to calculate stride, skip, and span. \( \Delta b \) is calculated by subtracting the \( b \) values of two subsequent MWMs from each other and \( \Delta a \) is calculated accordingly. When using an 128-bit access, all 16 pixels (1 Byte per pixel) of a row of SAD16×16 can be accessed in one MWM which leads to a value of \( \Delta b = 0 \) as shown in the table. Thus, stride is 0 and skip is always used to access the next address (i.e. \( \text{scan}\!= 1 \)). Note that this situation could also be handled by only using the stride value of the AGU and configuring scan and skip to 0, but due to our normalization and automatic detection, our systems relies on the skip value in such scenarios. The determined skip value is calculated from \( \Delta a \) and the variable value of \( x \), i.e. to calculate the next address, the value \( 4z \) has to be added.

A more interesting scenario is shown for memops 6–8 in Table 1 when considering an LSU bitwidth of \( W_{LSU} = 64 \).
Table 1: Excerpt of detected byte-offsets in SAD16×16, showing how the AGU parameters are extracted

<table>
<thead>
<tr>
<th>compiler output</th>
<th>128-bit MWM</th>
<th>(\Delta a)</th>
<th>(\Delta b)</th>
<th>64-bit MWM</th>
<th>(\Delta a)</th>
<th>(\Delta b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 base+0xc+0</td>
<td>#1</td>
<td>#1</td>
<td></td>
<td>#1</td>
<td>#1</td>
<td></td>
</tr>
<tr>
<td>1 base+0xc+4</td>
<td>#1</td>
<td>#1</td>
<td></td>
<td>#2</td>
<td>-</td>
<td>8</td>
</tr>
<tr>
<td>2 base+0xc+8</td>
<td>#1</td>
<td></td>
<td></td>
<td>#2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3 base+0xc+12</td>
<td>#1</td>
<td>#2</td>
<td></td>
<td>#2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>4 base+0x+0</td>
<td>#2</td>
<td>4</td>
<td>0</td>
<td>#3</td>
<td>#4</td>
<td>-</td>
</tr>
<tr>
<td>5 base+0x+4</td>
<td>#2</td>
<td>#3</td>
<td></td>
<td>#4</td>
<td>#4</td>
<td>-</td>
</tr>
<tr>
<td>6 base+0x+8</td>
<td>#2</td>
<td></td>
<td></td>
<td>#4</td>
<td>#4</td>
<td>-</td>
</tr>
<tr>
<td>7 base+0x+12</td>
<td>#2</td>
<td></td>
<td></td>
<td>#4</td>
<td>#4</td>
<td>-</td>
</tr>
<tr>
<td>8 base+8x+0</td>
<td>#3</td>
<td>#5</td>
<td>4</td>
<td>#5</td>
<td>#5</td>
<td>-</td>
</tr>
<tr>
<td>9 base+8x+4</td>
<td>#3</td>
<td></td>
<td></td>
<td>#6</td>
<td>-</td>
<td>8</td>
</tr>
<tr>
<td>10 base+8x+8</td>
<td>#3</td>
<td></td>
<td></td>
<td>#6</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>11 base+8x+12</td>
<td>#3</td>
<td></td>
<td></td>
<td>#6</td>
<td></td>
<td>8</td>
</tr>
</tbody>
</table>

(columns 6–8). Here, the second MWM (represented by memop 2) shows a stride value of \(\Delta b = 8\). The third MWM (represented by memop 4) actually has a b value of 0 which would lead to a negative \(\Delta b = 0 – 8\). This is detected by the system as indication that no longer stride has to be added, but skip. This also determines the span value of 2, i.e. after every 2 MWMs, skip has to be added. The value skip is again calculated from \(\Delta a\) and the variable value of \(x\).

There are some special cases where the first calculated delta value does not match the subsequent ones. For instance, when some values that actually do not belong to the memory stream are saved using the same base address. When considering a struct that contains some parameters and a subsequent array, then all elements have the base address of the struct, but the accesses to the parameters do not follow the pattern for the array accesses. For those cases the first MWM of the list is dropped from the bucket and the algorithm is restarted. The dropped element is either transformed into its own memory stream or excluded from the CI (depending on the number of available AGUs).

6. RESULTS

For our implementation we chose libFirm [25], a compiler framework based on the static single assignment (SSA) principle throughout all phases of compilation and optimization. Its intermediate representation (IR) consists of a combined control- and dataflow graph (CDFG) and an additional graph which connects all memory operations, thus modeling memory in SSA fashion, too. Other IRs without such a memory dependence graph can be easily extended by iterating over all nodes and linking memory operations encountered in the process. libFirm is built with extensibility in mind, this allows us to easily include our code as separate compilation phase and execute it for every IR graph after optimizations are done.

6.1 Experimental Setup

The targeted hardware architecture is a reconfigurable processor with \(N_{inR} = 4\) register read ports, \(N_{outR} = 2\) register write ports, 2 LSUs with a bitwidth of \(W_{LSU} = 128\) each, and and \(N_{AGU} = 4\) AGUs. The compiler was run in Linux on a DualCore i3 running at 3.10 GHz. A SystemC model of a SPARCv8 CPU was used to estimate the amount of cycles for a kernel. Code coverage is calculated from the cycles needed within the identified CI compared to the cycles needed for the whole kernel.

6.2 Benchmark Kernels

The selected benchmarks listed in Table 2 are typical kernels from multimedia applications which are used in streaming applications, like video playback and video encoding. They were selected to represent typical code sections that benefit from memory stream merging and hardware acceleration. DCT4×4 is the kernel of the ‘discrete cosine transformation’ that operates on a 4×4 32-bit sub-array of a larger array (which contains the entire image) and that stores the result in a separate array. SAD16×16 calculates the ‘sum of absolute differences’ and is used in motion estimation algorithms of video encoders to estimate the motion from a 16×16 region of the image relative to another image. It reads two 16×16 8-bit sub-arrays (from the two images that are compared) and calculates a single 32-bit value that is written to the register file. SATD4×4 is similar to SAD16×16 and also used in motion estimation. Hadamard4×4 is another transformation that is used in video encoding. Like SATD4×4 and DCT4×4 it operates on a 32-bit sub-array of a larger array. Matrix16×16 is the inner kernel of a matrix multiplication and is called several times to multiply two larger square matrices. GSMEncode is taken from the MiBench benchmark suite [26]. It is a streaming audio compression algorithm used in mobile phones that compresses frames of 160 16-bit samples into 33-byte frames.

The smallest kernels DCT4×4, SATD4×4, Hadamard4×4 and GSM only consist of the inner loop whereas Matrix16×16 still contained one loop which was to be unrolled during compilation. Those two test programs were compiled with different loop unrolling factors to show how resulting memory access patterns differ depending on how many memory operations are available in the loop body. The maximum unroll factor stems from how often the loop is repeated, e.g. for the Matrix Multiplication the maximum is sixteen as the inner loop operates on one line which is 16 elements wide.

6.3 Custom Instruction Analysis

Our extended compiler successfully identifies CIs for all benchmarks. Due to the memory operations threshold \(N_{ldst}\) in Algorithm 1 it is not necessary to select a specific basic block within the benchmark. Tables 3 and 4 show the results obtained by compiling the benchmarks. The ‘Unroll factor’
Table 3: Identified custom instruction for each benchmark; all values are [nodes per CI]

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>CI size [instr.]</th>
<th># load instr.</th>
<th>orig merged reduction [%]</th>
<th># store instr.</th>
<th>orig merged reduction [%]</th>
<th># cycles &amp; cover.</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCT4×4</td>
<td>325</td>
<td>8</td>
<td>5</td>
<td>38%</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>SATD4×4</td>
<td>303</td>
<td>8</td>
<td>2</td>
<td>75%</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Hadamard4×4</td>
<td>244</td>
<td>16</td>
<td>16</td>
<td>0%</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>GSMEncode</td>
<td>380</td>
<td>99</td>
<td>14</td>
<td>86%</td>
<td>33</td>
<td>3</td>
</tr>
<tr>
<td>SAD16×16</td>
<td>2701</td>
<td>128</td>
<td>32</td>
<td>75%</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Matrix16×16u16</td>
<td>2625</td>
<td>256</td>
<td>64</td>
<td>75%</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>Matrix16×16u8</td>
<td>1403</td>
<td>128</td>
<td>32</td>
<td>75%</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Matrix16×16u0</td>
<td>225</td>
<td>32</td>
<td>32</td>
<td>0%</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4: Compiler runtime and introduced overhead for CI identification

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>total compiler runtime [ms]</th>
<th>introduced overhead</th>
<th>selected parts of algorithm in [%]</th>
<th>normalize</th>
<th>extract</th>
<th>mwMerge</th>
<th>convex</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCT4×4</td>
<td>11</td>
<td>4%</td>
<td>0.4%</td>
<td>0.08%</td>
<td>25.2%</td>
<td>21.2%</td>
<td></td>
</tr>
<tr>
<td>SATD4×4</td>
<td>13</td>
<td>35%</td>
<td>0.3%</td>
<td>0.08%</td>
<td>15.3%</td>
<td>19.1%</td>
<td></td>
</tr>
<tr>
<td>Hadamard4×4</td>
<td>9</td>
<td>82%</td>
<td>0.3%</td>
<td>0.30%</td>
<td>80.1%</td>
<td>1.2%</td>
<td></td>
</tr>
<tr>
<td>GSMEncode</td>
<td>80</td>
<td>43%</td>
<td>0.8%</td>
<td>0.13%</td>
<td>33.0%</td>
<td>9.4%</td>
<td></td>
</tr>
<tr>
<td>SAD16×16</td>
<td>1053</td>
<td>93%</td>
<td>72.0%</td>
<td>0.05%</td>
<td>19.0%</td>
<td>2.1%</td>
<td></td>
</tr>
<tr>
<td>Matrix16×16u16</td>
<td>2578</td>
<td>97%</td>
<td>74.0%</td>
<td>0.09%</td>
<td>21.2%</td>
<td>1.4%</td>
<td></td>
</tr>
<tr>
<td>Matrix16×16u8</td>
<td>663</td>
<td>93%</td>
<td>66.0%</td>
<td>0.08%</td>
<td>24.4%</td>
<td>2.8%</td>
<td></td>
</tr>
<tr>
<td>Matrix16×16u0</td>
<td>249</td>
<td>88%</td>
<td>59.7%</td>
<td>0.08%</td>
<td>24.3%</td>
<td>4.7%</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6: Benchmark of state-of-the-art enumeration algorithm [8]; result is for SATD4×4 using 1–4 read ports and 3 write ports

Table 3: Identified custom instruction for each benchmark; all values are [nodes per CI]

1 The suffix XuY denotes that out of X iterations, Y were unrolled; the rest was executed as loop

Table 4: Compiler runtime and introduced overhead for CI identification

Benchmark     | total compiler runtime [ms] | introduced overhead | selected parts of algorithm in [%] | normalize | extract | mwMerge | convex |
---------------|-----------------------------|---------------------|-----------------------------------|-----------|---------|---------|--------|
DCT4×4        | 11                          | 4%                  | 0.4%                              | 0.08%     | 25.2%   | 21.2%   |
SATD4×4       | 13                          | 35%                 | 0.3%                              | 0.08%     | 15.3%   | 19.1%   |
Hadamard4×4   | 9                           | 82%                 | 0.3%                              | 0.30%     | 80.1%   | 1.2%    |
GSMEncode     | 80                          | 43%                 | 0.8%                              | 0.13%     | 33.0%   | 9.4%    |
SAD16×16      | 1053                        | 93%                 | 72.0%                             | 0.05%     | 19.0%   | 2.1%    |
Matrix16×16u16 | 2578                        | 97%                 | 74.0%                             | 0.09%     | 21.2%   | 1.4%    |
Matrix16×16u8 | 663                         | 93%                 | 66.0%                             | 0.08%     | 24.4%   | 2.8%    |
Matrix16×16u0 | 249                         | 88%                 | 59.7%                             | 0.08%     | 24.3%   | 4.7%    |

Figure 6: Benchmark of state-of-the-art enumeration algorithm [8]; result is for SATD4×4 using 1–4 read ports and 3 write ports

u in the matrix multiplication benchmark denotes how often the inner loop of this kernel was unrolled before the CI identification process was started.

Table 3 shows statistics of the identified custom instruction; the numbers denote the amount of IR nodes per CI. Our greedy memop merging approach works very well as can be particularly seen in Matrix16×16u16, Matrix16×16u8 and SAD16×16. All available (32-bit) loads were correctly merged to multi-word loads which are filled to the maximum memory width $W_{LSU}$ of the targeted architecture. In GSMEncode the effect is extremely visible because the algorithm processes 8-bit values so that the algorithm could merge up to 16 memops into one MWM. The generated CIs contain 999 basic operations on average and the maximum size our algorithm achieved was 2701 nodes that could be converted in one CI.

In Table 4 the total runtime of the compiler is shown along with the overhead introduced by the CI generation algorithm. The table also shows a breakdown of the needed time in percent of the total runtime for the major algorithm steps. We can see that our algorithm runtime stays within reasonable limits even for very large basic blocks / CIs like the Matrix16×16u16 and the SAD16×16 adding a maximum of 2.5 seconds to the absolute compile time. The extract step of the algorithm consistently has only very small impact on the overall time, whereas the effort for the normalization step varies between benchmarks. Compared to state-of-the-art algorithms like [8], which generate up to millions of candidates as seen in Figure 6, our approach does not need the candidate selection step. All CIs are convex and can be executed without overlapping due to the way a CI is constructed by our algorithm.

The presented algorithm produces very large CIs due to the aggressive loop unrolling. If a CI becomes too large for implementation in hardware, the unroll factor can be reduced. The effect of different unroll factors is analyzed with the Matrix16×16 example. Here we forced the compiler to use 3 different unroll factors: u16, u8 and u0. When fully unrolled (u16), no loop remains and the memory operations can be merged best achieving an reduction of load instructions by 75%. The u8 case achieves the same reduction in load instructions while reducing the CI instruction size by 47%. Without loop unrolling the CI size shrinks further but there is no benefit from merged memory operations anymore due to the nonlinear memory read pattern. Coincidentally, the part of the kernel covered by the CI goes down to only 5% in contrast to the half and fully unrolled loops, which shows a code coverage that is linear dependent on the unroll factor. Except for the unmodified Matrix16×16u0 all kernels are exceptional well covered by the identified CI. For the Matrix16×16u16 only the innermost part of the inner loop can be transformed into a CI and none of the remaining arithmetic operations of the outer loops would be accelerated. A steady increase in coverage is seen with the increased unrolling up to the point were only minor application initialization before the computational kernel are not part of the CI. This holds for all examples except for the Hadamard4×4 where only 48% can be covered. This is also
reflected in the load instructions, which could not be reduced for Hadamard $4 \times 4$.

The low reduction of store instructions of most examples can be explained by the small amount of store nodes in the code, i.e. a single memory operation cannot be optimized. For the benchmarks with many store operations similar reductions than for the load instructions are achieved. A notable exception is the matrix multiplication, as the CI calculates the result for the first column of the matrix and thus does not store to consecutive memory locations.

7. CONCLUSION

This paper presents an algorithm to generate large custom instructions (CIs) that support memory access. It normalizes the memory address calculations and extracts the access patterns to initialize dedicated address generation units that are used by CIs to access main memory. In comparison to state-of-the-art enumeration algorithms that exclude memory access and commonly generate up to millions of candidates, we identify one large CI per basic block (typically covering the entire unrolled inner loop of a computational kernel) and thus eliminate the necessity for an additional selection step.

The high code coverage on those benchmarks shows that relevant code sections were correctly identified and the improvement rate directly indicates savings in execution time by using the identified CI.

The generated CIs are convex by the way they are constructed and do not overlap with each other, which means that each SI executes atomically and all CIs identified in one application can be used. Identification and merging of memory operations allows us to map multiple memory accesses to a single bulk access and thus reduce the number of memory accesses. By including memory operations into CIs, our approach realizes automatic compiler support for a new category of memory-bound accelerators (i.e. automatic creation of such CIs) which is not yet supported by state-of-the-art.

8. ACKNOWLEDGMENTS

This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center “Invasive Computing” (SFB/TR 89).

References