



ASP-DAC 2008



Enabling Run-Time Memory Data Transfer Optimizations at the System Level with Automated Extraction of Embedded Software Metadata Information

A. Bartzas¹, M. Peón Quirós², S. Mamagkakis³,
F. Catthoor^{3,4}, D. Soudris¹, J. M. Mendías²

¹ VLSI Design & Testing Center, Democritus Univ. Thrace, Greece

² DACYA, Univ. Complutense de Madrid, Spain

³ IMEC vzw., Belgium

⁴ Also Prof. at Katholieke Univ. Leuven, Belgium



Supported by:
PENED 03ED593 (Greece)
TIN 2005-5619 (Spain)
HPMT-CT-2000-00031 (E.U.)



MARIE CURIE ACTIONS

Introduction – Motivation



New **Dynamic Multi-threaded** Applications



Limited Resources = Need for optimized DMA usage

- Analyze concurrent accesses to DRAMs
 - **Multi-threaded** embedded network system
 - Use of **real wireless input traces**
- Allow early stage identification of block transfers of **Dynamic Data Types** (e.g., linked lists) from source code
- Derive optimization for DMA usage and design from **Software** and **Hardware Metadata**

Available DMA Options

- Typical use:
 - Programmer writes loops to process data from arrays
 - Programmer identifies potential block transfers
 - To move data in the memory hierarchy
 - To transfer data between buffers (processing or I/O).
 - Block transfer primitives are applied manually (`memcpy` or DMA calls)
- We propose:
 - **Tool-flow** to automatically identify potential block transfers of Dynamic Data Types in early design stages
 - **Tuning Software Metadata:** adjust minimum block transfer size that is done with the DMA
 - **Parameterizing Hardware Metadata:** maximum number of cycles that the DMA can keep the bus without being interrupted by the processor

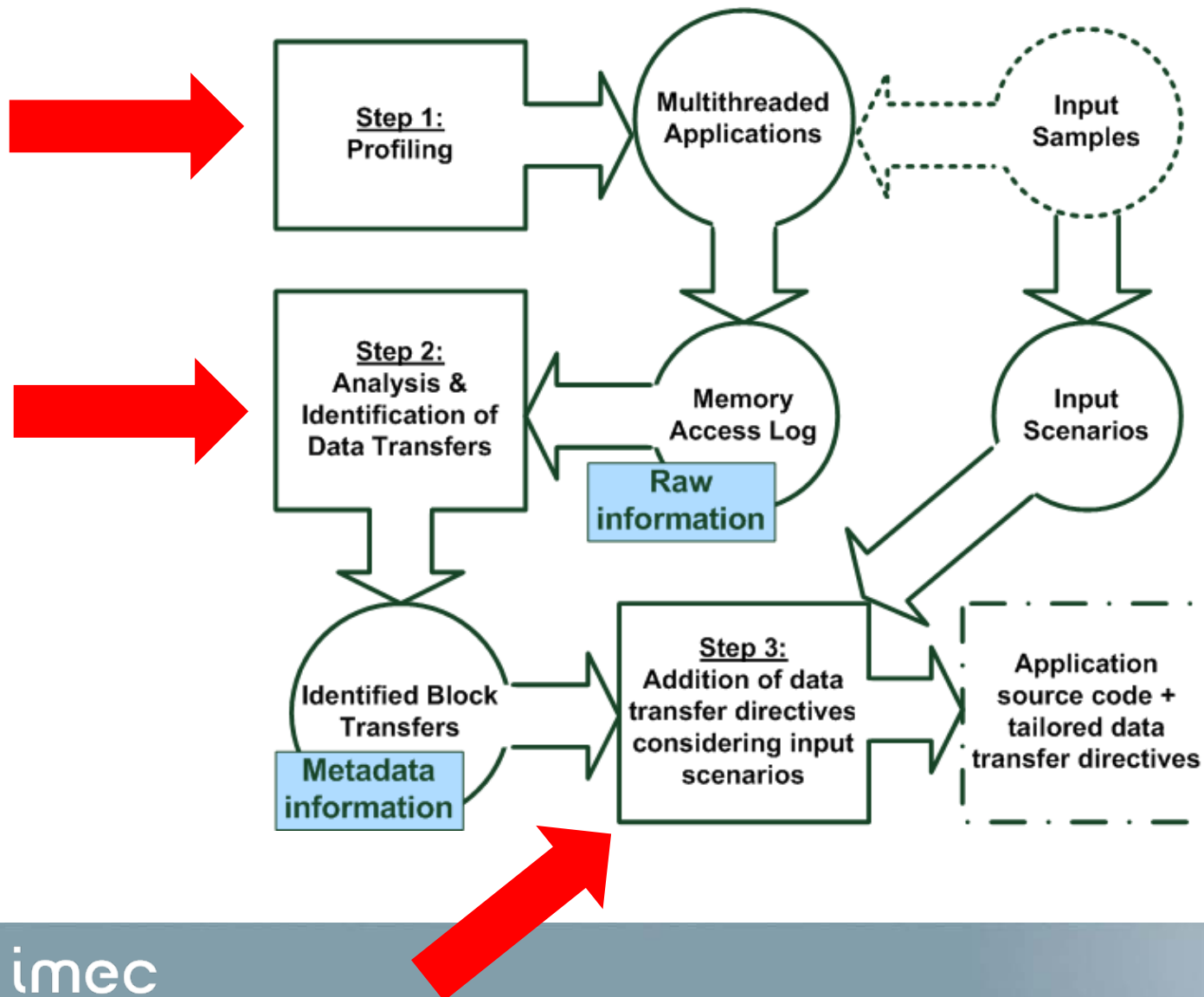
Presentation Overview

- Introduction – Motivation
- Software Metadata
- Methodology Overview
- Experimental Results
- Conclusions – Future Work

What is *Software Metadata*?

- We define as *Software Metadata* the representation of the characteristics of dynamic applications in respect of:
 - Requested memory footprint
 - Data access behaviour
- ... whereas *Hardware Metadata* describe the characteristics of the platform
 - Allowing easily design, test and verify hardware platforms [SPIRIT Consortium – IP-XACT]

Methodology Overview



Step1: Raw Information Extraction

- Profile the dynamic applications
 - Using as many realistic inputs as possible/within time limits, etc.
- How do we do it? → using Profiling Templates
 - Manual, typed-based approach
 - Each variable type (for the variable we are interested in) is annotated
 - The propagation of the type-change is performed by the compiler
 - Windows + Linux
 - 2-3x slower execution time

What information do we capture?

- Allocation and de-allocation of the dynamic memory, identified by the specific variable in the application
- Dynamic memory accesses (reads and writes) identified by the specific variable in the application
- Operations on dynamic data types, identified by the specific data type in question
- Control-flow paths that lead to the locations where these operations are being done
- Thread identifiers within which the aforementioned operations occur

Step2: Analysis and Identification of Data

Algorithm 1 Data transfers identification

```
1: function TRANSFERIDENTIFICATION
2:   aliveBlocks : List of Blocks
3:   recordOfTransfers : List of Transfers
4:   for all event  $\in$  logFile do
5:     if event is allocation then
6:       aliveBlocks.Insert(new Block(address, size, dataTypeID))
7:     else if event is deallocation then
8:       block  $\leftarrow$  aliveBlocks.FindBlock(address)
9:       if IsValid(block.activeTransfer) then
10:        recordOfTransfers.Insert(block.activeTransfer)
11:       end if
12:       aliveBlocks.DeleteBlock(address)
13:       delete block
14:     else if event is memory access then
15:       block  $\leftarrow$  aliveBlocks.FindBlock(address)
16:       transfer  $\leftarrow$  block.activeTransfer
17:       if IsValid(block) and IsValid(transfer) then
18:         if IsConsecutive(transfer, address, threadID, direction)
19:           then
20:             transfer.Update(address)
21:           else
22:             recordOfTransfers.Insert(transfer)
23:             block.activeTransfer  $\leftarrow$  new Transfer
24:               (address, dataTypeID, threadID, direction)
25:             end if
26:           else
27:             block.activeTransfer  $\leftarrow$  new Transfer
28:               (address, dataTypeID, threadID, direction)
29:             end if
30:           end if
31:         end if
32:       end for
33: end function
```

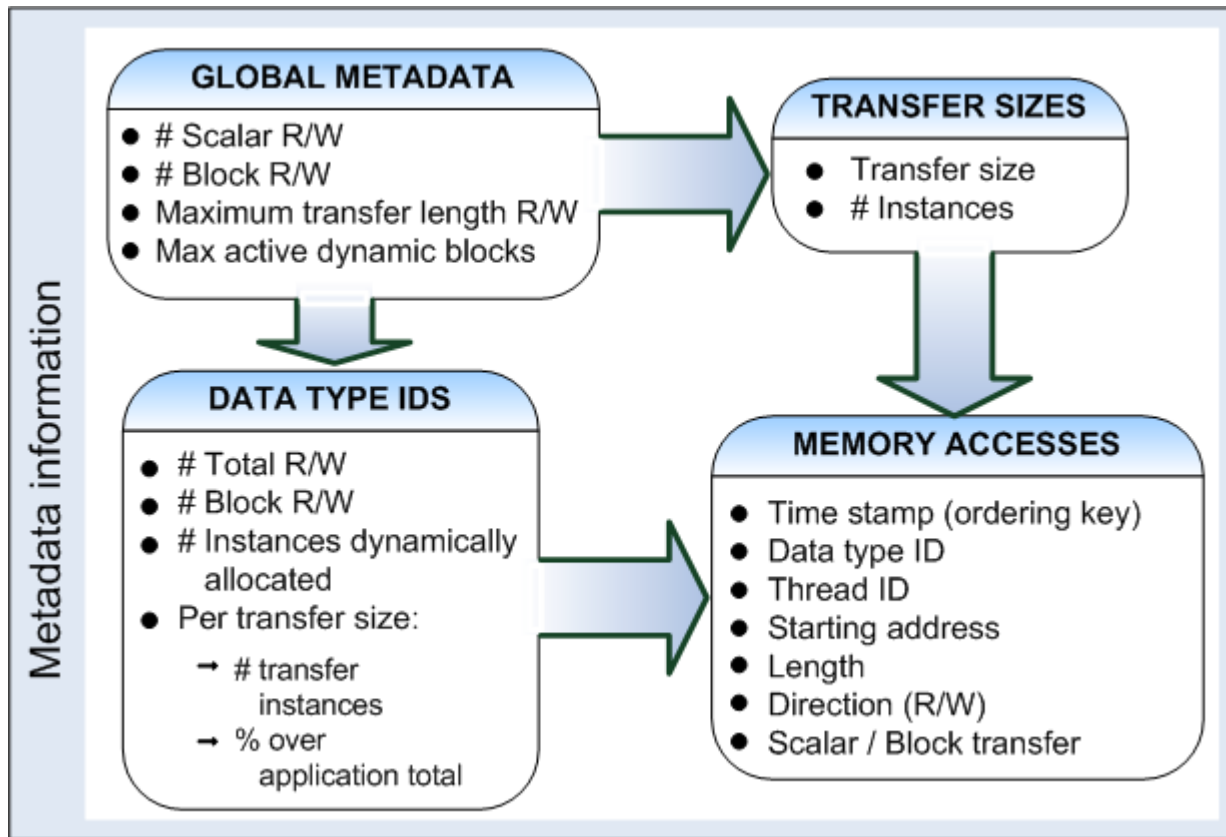
Step2: Analysis and Identification of Data Transfers

- We read ALL the data transfers captured in the log file
- We capture ALL allocation deallocation events
- We pack together consecutive memory accesses (to the same data – from the same thread) as a block transfer
- We go from independent, interleaved accesses, to consolidated block transfers.

Algorithm 1 Data transfers identification

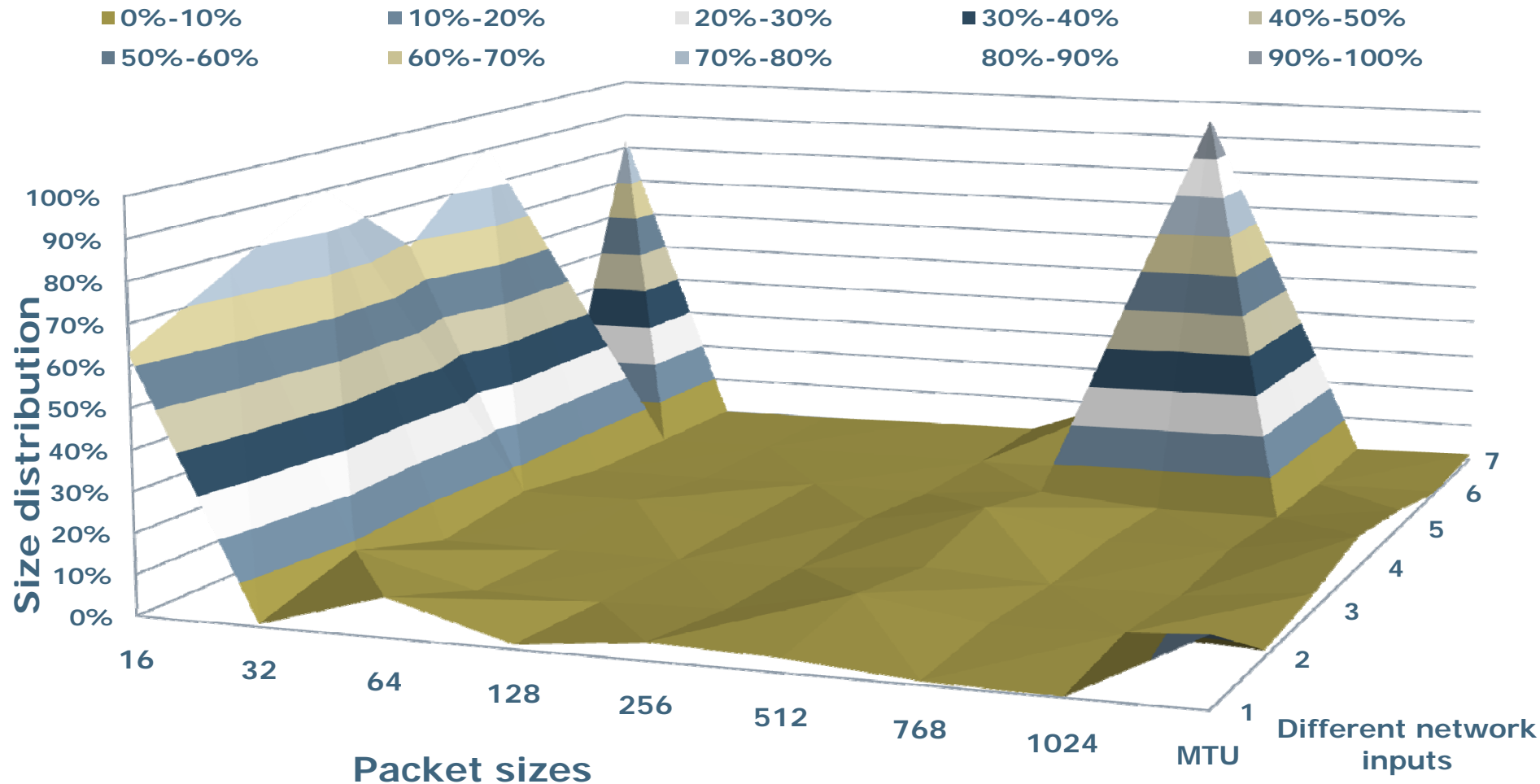
```
1: function TRANSFERIDENTIFICATION
2:   aliveBlocks : List of Blocks
3:   recordOfTransfers : List of Transfers
4:   for all event  $\in$  logFile do
5:     if event is allocation then
6:       aliveBlocks.Insert(new Block(address, size, dataTypeID))
7:     else if event is deallocation then
8:       block  $\leftarrow$  aliveBlocks.FindBlock(address)
9:       if IsValid(block.activeTransfer) then
10:        recordOfTransfers.Insert(block.activeTransfer)
11:       end if
12:       aliveBlocks.DeleteBlock(address)
13:       delete block
14:     else if event is memory access then
15:       block  $\leftarrow$  aliveBlocks.FindBlock(address)
16:       transfer  $\leftarrow$  block.activeTransfer
17:       if IsValid(block) and IsValid(transfer) then
18:         if IsConsecutive(transfer, address, threadID, direction)
19:           then
20:             transfer.Update(address)
21:           else
22:             recordOfTransfers.Insert(transfer)
23:             block.activeTransfer  $\leftarrow$  new Transfer
24:               (address, dataTypeID, threadID, direction)
25:           end if
26:         else
27:           block.activeTransfer  $\leftarrow$  new Transfer
28:             (address, dataTypeID, threadID, direction)
29:         end if
30:       end if
31:     end for
32:   end function
```

Software Metadata Structure

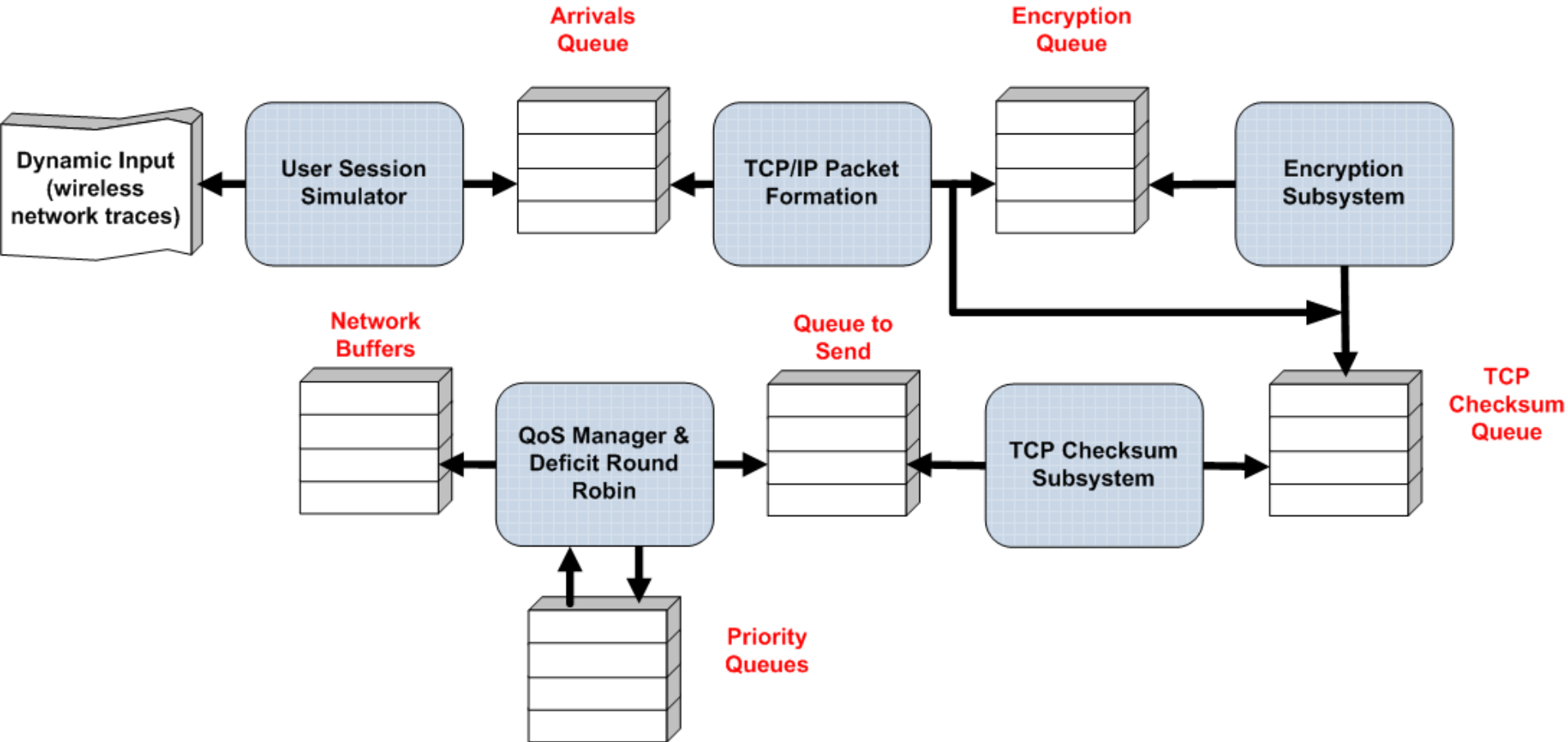


Step3: Evaluation of SW Metadata

Packet size distribution among different inputs

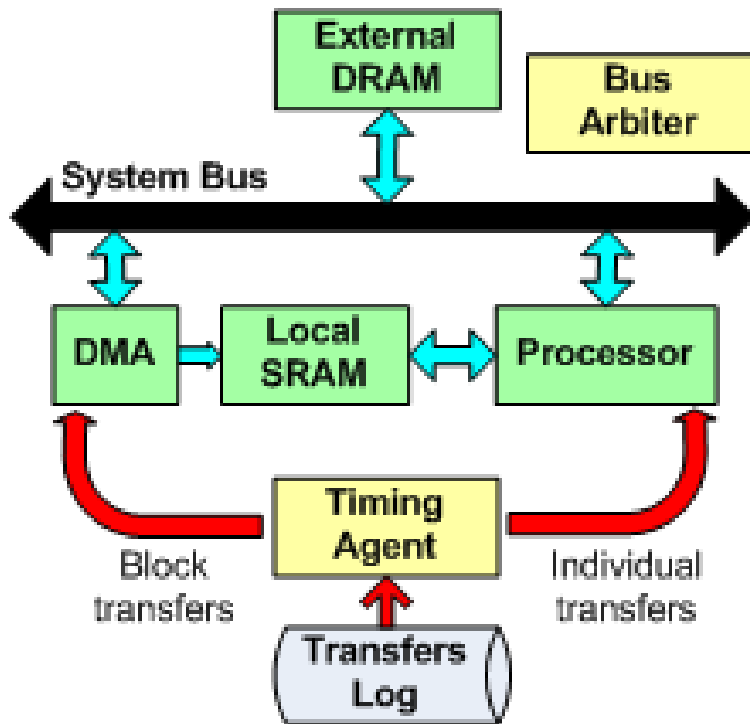


Software Testbench



- 5 threads
- Communication through synchronization queues
- Real network traces

Simulator Architecture



- Use of a memory hierarchy simulator to evaluate our approach
- Simulates:
 - Concurrent behavior of processor and DMA
 - Based on trace of memory accesses generated using the original profiling information
- Results:
 - Number of row activations in DRAM banks
 - Total energy consumption of memory subsystem
 - Mean latency of memory hierarchy

Usage of Software Metadata

Algorithm 2 Optimized data transfer function according to software Metadata.

```
1: function TRANSFERCOPY(source, destination, dataTypeID, size)
2:   if GlobalScenario = ScenarioA then ▷ Perform a software copy
3:     memcpy(source, destination, size)
4:   else if (GlobalScenario = ScenarioB) and
5:     (dataTypeID = X) then ▷ Do locked DMA transfer.
6:     DMATransf(source, destination, size, WAIT)
7:   else if ... then
8:     ...
9:   end if
10: end function
11: function PROCESSDATA(input, output, size)
12:   TransferCopy(input, SCRATCHPAD, dataType, size)
13:   DoComplexProcessing(SCRATCHPAD, size)
14:   TransferCopy(SCRATCHPAD, output, dataType, size)
15: end function
```

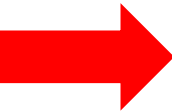
Only for the dynamic data that is profitable use the DMA to perform a data transfer

- The DMA is regulated by the Software Metadata extracted by our tools

In the client function, the invocation of TransferCopy leads to implementation of the transfer policy, according to the identified case + the SW metadata of the original application allowed us to identify the relevant block transfers from the original source code.

Experimental Results _(1/5)

TABLE I
GLOBAL STATISTICS FOR IDENTIFIED BLOCK TRANSFERS.

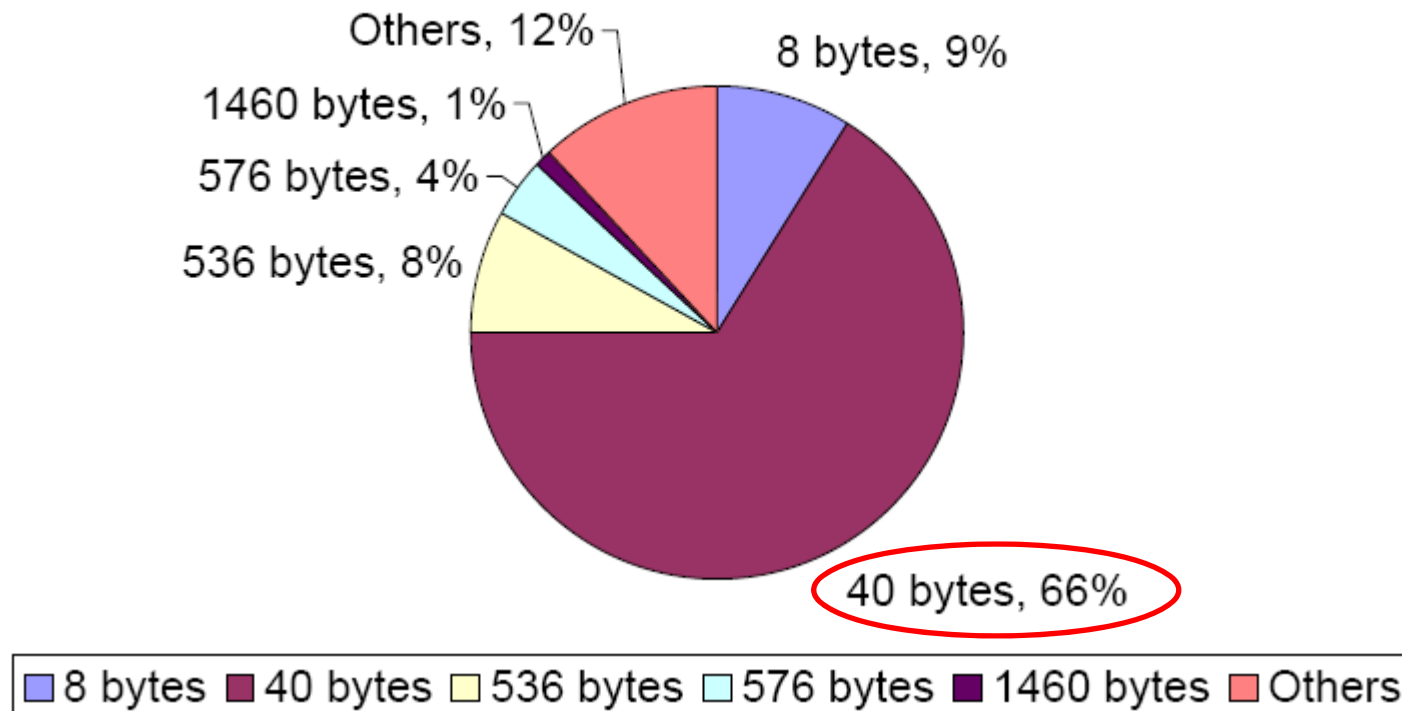


Input	# Read transfers	# Write transfers	Max Transfer length	Mean Transfer length	Most active data type ID
01	396,347	79,016	1,500	203	2
02	40,663	37,568	1,500	109	2
03	250,283	213,094	1,500	107	14
04	294,408	313,144	1,500	181	2
05	299,346	285,951	1,300	108	14
06	254,012	253,248	576	344	2
07	2,641,875	371,245	1,500	87	2

Highest abstraction layer
Create statistics and identify block transfers for each input

Experimental Results (2/5)

Percentage distribution of data transfer lengths



Second level of abstraction

Extract the frequency of data transfers, according to their size

Experimental Results (3/5)

TABLE II
DISTRIBUTION OF TRANSFER SIZES FOR EACH DYNAMIC DATA TYPE ID,
AVERAGED FOR ALL INPUTS.

Percentage over total	Transfer size	# Transfers	ID
55%	40	382,740	01
18%	40	126,078	14
7%	8	48,534	02
6%	536	63,210	02
5%	576	31,605	14
2%	8	16,940	01
1%	1460	6,182	02
1%	468	3,538	02
2%	Others	17,501	Others

Details for the concrete dynamic data type IDs associated with each transfer length

Experimental Results (4/5)

TABLE III

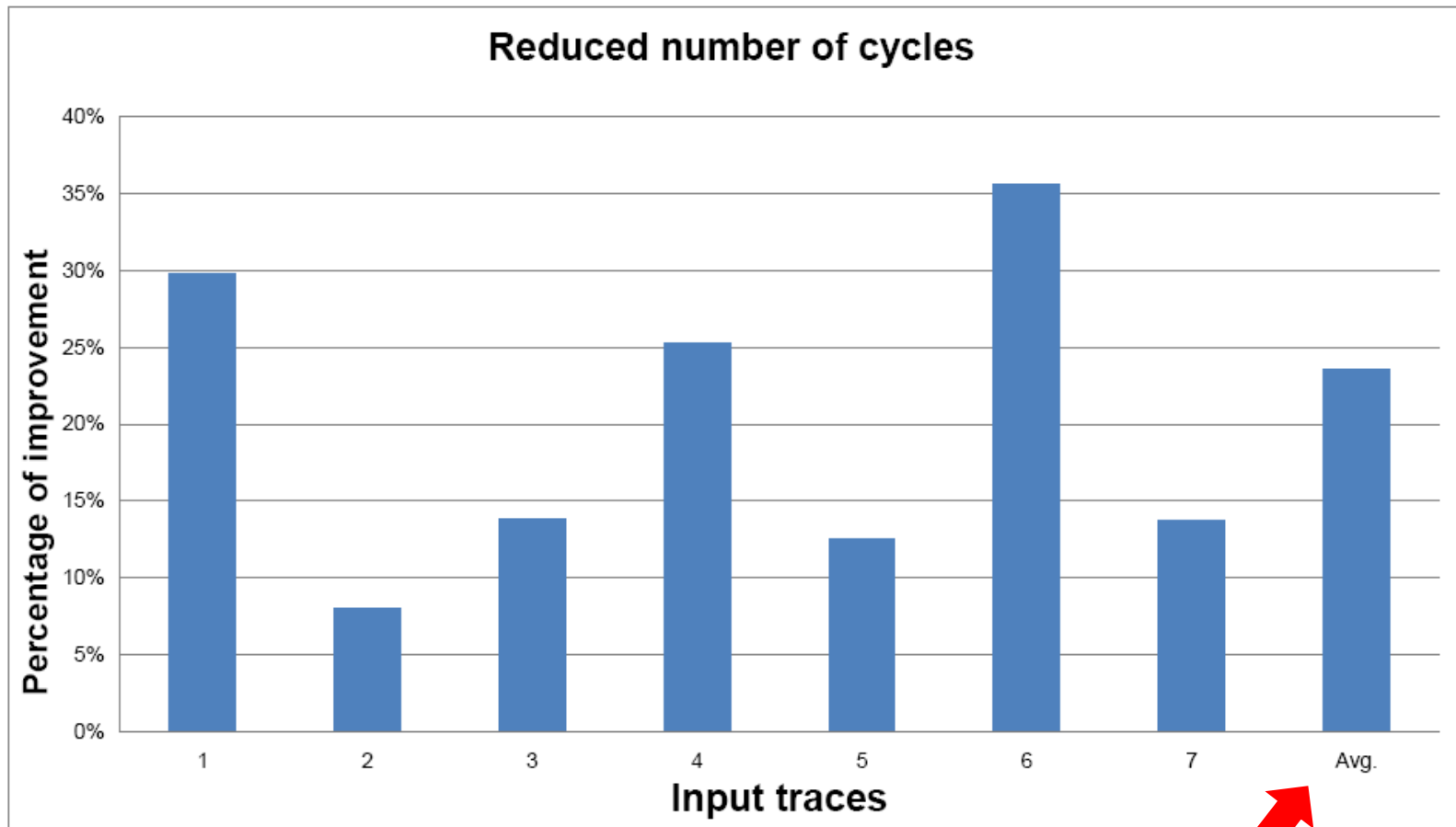
REDUCTION OF THE TOTAL NUMBER OF EXECUTED CYCLES WITH DMA-BASED OPTIMIZATIONS EXPLOITING OUR EXTRACTED METADATA.

Input	# Cycles proc (with DMA)	# Cycles DMA	# Cycles (with DMA)	# Cycles (without DMA)
01	688,552	2,564,269	3,252,821	4,634,128
02	369,934	628,622	998,556	1,085,164
03	1,665,311	3,177,720	4,843,031	5,607,312
04	2,875,726	8,446,861	11,322,587	15,145,873
05	2,300,094	4,282,901	6,582,995	7,527,066
06	1,420,904	9,025,292	10,446,196	16,222,819
07	3,023,595	5,404,975	8,428,570	9,775,976
Avg.	1,763,445	4,790,091	6,553,536	8,571,191



System cycles from the memory point of view

Experimental Results (5/5)



Reduction in memory subsystem cycles from 8% up to 35.6% and on average 23.56% for all the traces

Conclusions and Future Work

- We have presented a framework that extracts SW metadata...
- ...enabling memory data transfer optimizations
- Successful identification of block transfers
 - Evaluation using real network traces
 - 23.5% reduction of memory cycles
- Future work:
 - Extensions towards data transfer scheduling
 - Enhanced block transfer identification
 - Improved high-level simulator

Thank You! Questions?

Contact: Alexandros Bartzas
ampartza@ee.duth.gr



aspire invent achieve

