Wydajność architektury STM32 w zakresie wykonywania kodu pośredniego dla systemów sterowania

Marcin Hubacz, Jan Sadolewski, Bartosz Trybus
Politechnika Rzeszowska, Wydział Elektrotechniki i Informatyki, ul. Wincentego Pola 2, 35-021 Rzeszów

Streszczenie: W artykule przedstawiono badania wydajności wykonywania przez mikrokontrolery STM32 kodu dla maszyny wirtualnej (tzw. kodu pośredniego) dedykowanej dla systemów sterowania. Architektura ARM zastosowana w tych układach odznacza się ograniczeniami związanymi z dostępem do niewyrównanych adresów. Zaproponowano trzy sposoby wyeliminowania tych ograniczeń, a każdy z nich poddano zestawowi testów mających ustalić ich wydajność. Testy przeprowadzono dla dwóch trybów działania, tj. z 16- i 32-bitowym adresowaniem dla różnych generacji układów. Wyniki testów pozwalają dobrać właściwe rozwiązanie dla określonej platformy.

Słowa kluczowe: STM32, ARM, maszyna wirtualna, kod pośredni

1. Wprowadzenie

Na budowę typowego rozwiązania składa się sprzęt oraz oprogramowanie. Nowe technologie, ukierunkowane na uniwersalne rozwiązania sprawiają, że tworzone oprogramowanie w dużej mierze traci idee optymalizacji i wydajności na rzecz przenaszalności i łatwego wdrożenia. Dotyczy to również systemów sterowania.

Od strony sprzętowej sterowniki PLC konstruuje się w dużej mierze o oparciu o mikrokontroler. Od jego właściwości zależą możliwości oferowane przez system sterowania, w tym skalowalność. Aktualnie dostępne nawet małe mikrokontrolery mają spore możliwości obliczeniowe, obsługują dużą ilość pamięci operacyjnej, co czyni je częstym wyborem w zastosowaniach wielu gałęzi elektroniki. Coraz chętniej wykorzystywane są przede wszystkim układy oparte o architekturze RISC, w szczególności ARM. Jednym z przedstawicieli jest popularna rodzina mikrokontrolerów STM32 firmy STMicroelectronics, która obejmuje układy spełniające różne wymagania [3]. W jej skład wchodzą mikrokontrolery ogólnego zastosowania, nakierowane na dobrą wydajność oraz charakteryzujące się niskim zużyciem energii. Przy wyborze jednego z układów tej platformy można brać pod uwagę takie czynniki, jak wydajność, cena, zasób peryferii oraz portów GPIO.

Przenaszalność oprogramowania między poszczególnymi przedstawicielami tej rodziny może jednak sprawiać problemy. Występują bowiem istotne różnice miedzy architekturą rdzeni procesorów. Różnice te wpływają na wydajność oprogramowania i mogą spowodować błędy w jego działaniu. W szczególności może to dotyczyć oprogramowania sterującego wykonywanego po stronie sterownika w formie kodu pośredniego.

Autor korespondujący:
Marcin Hubacz, m.hubacz@prz.edu.pl

Artykuł recenzowany
nadesłany 29.11.2020 r., przyjęty do druku 18.02.2021 r.

2. Kod pośredni

Efektem działania typowych kompilatorów języków programowania jest binarny kod maszynowy przeznaczony dla określonej platformy docelowej (*target*), której najważniejszym elementem jest architektura jednostki centralnej. Uwzględniany jest przy tym m.in. zestaw instrukcji procesora, rozmiar słowa, układ pamięci. Dzięki temu utworzony kod jest efektywnie uruchamiany i wykonywany na platformie docelowej. Kompilator wraz z programem łączącym (*linker*) może przy tym skorzystać z jej cech i utworzyć kod optymalny, którego wydajność będzie najlepsza dla tej platformy.

Wada tego podejścia objawia się w przypadku, gdy oprogramowanie ma być uruchamiane na sterownikach różnych pod względem architektury sprzętowej. Wówczas dla każdego z nich niezbędne jest wykorzystanie oddzielnego kompilatora, dedykowanego dla danej platformy (*cross-compiler*). Kod binarny jest następnie przenoszony na docelowe urządzenie i tam uruchamiany. Ze względu na różne implementacje kompilatorów, wynikowy kod może mieć inną charakterystykę czasową, a w niektórych przypadkach efekty działania programu po skompilowaniu mogą się różnić.

Rozwiązaniem alternatywnym jest użycie interpretera kodu po stronie sterownika. Interpreter działa w oparciu o kod źródłowy programu, a nie kod maszynowy przeznaczony dla docelowego procesora. Dzięki temu możliwe jest przygotowanie jednego kodu programu, który następnie jest umieszczany na różnych sterownikach. Kod źródłowy programu jest więc przenaszalny. W takim

przypadku interpreter przetwarza instrukcje w kodzie programu na odpowiednie działania realizowane za pomocą dostępnego sprzętu, przede wszystkim jednostki centralnej. Wadą tego rozwiązania jest czas potrzebny na interpretację kodu programu, przez co jego wykonywanie jest zwykle znacznie wolniejsze niż w przypadku przygotowywania kodu maszynowego. Dodatkowo, błędy w kodzie źródłowym są wykrywane dopiero podczas uruchomienia.

Rozwiązaniem kompromisowym, które stanowi podstawę niniejszej pracy jest zastosowanie tzw. maszyny wirtualnej. Polega ono na utworzeniu po stronie sterownika środowiska uruchomieniowego w formie emulowanego procesora wraz z dodatkowymi modułami niezbędnymi do wykonywania kodu. Kod ten jest przygotowany z uwzględnieniem charakterystyki wirtualnego procesora, jego zestawu instrukcji i emulowanego środowiska. Nie jest to już kod programu, lecz przygotowywany przez kompilator binarny kod maszynowy, przeznaczony dla wirtualnego procesora. Wystarczy więc jeden kompilator, a jego zastosowanie wiąże się z możliwością wykrycia błędów wcześniej, przed przesłaniem programu do sterownika. Kod binarny zwany jest kodem pośrednim (*intermediate code*), bowiem jego wykonywanie musi być realizowane przez maszynę wirtualną [6, 10, 14]. Jest to kod przenaszalny, uniwersalny, gdyż nie zależy od architektury platformy docelowej. Przygotowane w ten sposób moduły programowe mogą być dowolnie rozmieszczane w węzłach rozproszonego systemu sterowania, niezależnie od jednostek centralnych w tych węzłach. Wadą tego rozwiązania jest dłuższy czas wykonywania (choć nie tak długi jak przy interpreterze kodu źródłowego) oraz konieczność implementacji maszyny wirtualnej na każdym typie sterownika.

3. Środowisko badawcze

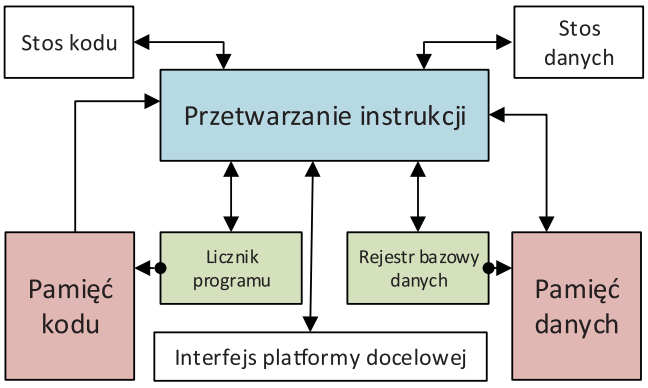
3.1. Maszyna wirtualna CPDev

Założeniem projektowym środowiska programistyczno-uruchomieniowym CPDev [15] są wcześniej wymienione przenaszalność i uniwersalność oprogramowania tworzonego w językach normy PN-EN 61131-3 [8, 13], osiągane przede wszystkim za pomocą kodu pośredniego. Głównym filarem pakietu jest maszyna wirtualna CPDev, którą można dostosować do określonej platformy sprzętowej [2]. Wykorzystanie wirtualizacji pozwoliło na oddzielenie architektury od wykonywanego kodu programu. Efektem ubocznym takiego rozwiązania jest zmniejszenie wydajności opracowywanego systemu. Maszyna wirtualna została zaimplementowana m.in. w sterownikach opartych na mikrokontrolerach, w tym ARM, x86/x64, Xtensa oraz na układach FPGA [7].

Rysunek 1 przedstawia najważniejsze moduły maszyny wirtualnej. Za wykonywanie kodu pośredniego odpowiada interpreter instrukcji. Pamięć kodu przechowuje kod pośredni programu, zaś pamięć danych – wartości zmiennych. Podczas wykonywania kodu i wywołania podprogramów używane są odpowiednie stosy.

Pierwotnie uwzględniono w maszynie wirtualnej 16-bitową adresację pamięci (*maszyna 16-bitowa*), co ograniczyło rozmiar pamięci kodu oraz danych do 64 kB. Środowisko CPDev było planowane do małych i średnich systemów sterowania wykorzystujących 8- i 16-bitowe mikrokontrolery, a więc wartość ta była w zupełności wystarczająca dla typowych zastosowań [11]. W późniejszym okresie użytkownicy zaczęli stosować CPDev do większych systemów, gdzie prym wiodą rozbudowane środowiska CODESYS, ISaGRAF itp. [1, 9]. Okazało się wówczas, że oprogramowanie sterujące może nie zmieścić się w dostępnym obszarze pamięci (64 kB). W szczególności problem ten objawił się przy konstruowaniu algorytmów dla autopilota okrętowego [16].

W celu rozwiązania tego problemu zwiększono rozmiar adresu maszyny wirtualnej do 32 bitów (maszyna 32-bitowa), co daje teoretyczny rozmiar kodu i danych do 4 GB. Rysunek 2 przedstawia wyniki kompilacji wyrażenia logicznego zapisanego w języku ST [12] (rys. 2a) i ustalającego stan zmiennej wyjściowej *MOTOR* na podstawie wartości zmiennych *START* i *STOP* oraz stanu wcześniejszego. Poniżej (b) przedstawiono wygenerowany przez kompilator kod asemblerowy używający mnemoników rozkazów maszyny wirtualnej. Są to rozkazy skoków (*JZ*, *JMP*), operatory logiczne (*OR*, *NOT*) oraz instrukcja inicjująca (*MCD*).



Rys. 1. Uproszczona architektura maszyny wirtualnej używanej do badań
Fig. 1. The simlified archiitecture of the virtual machine used for the research

We fragmentach (c) i (d) zamieszczono kod pośredni wykonywany przez maszynę wirtualną. Część (c) dotyczy kodu 16-bitowego, a (d) – 32-bitowego. Rozkazy są tu zastąpione ich kodami liczbowymi (np. *OR* – 0920), a zmienne i etykiety ich adresami w pamięci kodu i danych (np. *MOTOR* – 0200 i 02000000).

Niestety, wydajność dla niektórych konfiguracji sprzętowych maszyny 32-bitowej okazała się gorsza niż maszyny 16-bitowej,

a)
```
MOTOR:=(START OR MOTOR) AND NOT STOP
```

b)
```
 $DEFAULT.OR ?LR?ANDA000B, START, MOTOR
 $VMSYS.JZ ?LR?ANDA000B,
        :?Start_stop.MAIN?AND000A
 $DEFAULT.NOT ?LR?ANDA000D, STOP
 $VMSYS.JZ ?LR?ANDA000B,
        :?Start_stop.MAIN?AND000A
 $VMSYS.MCD MOTOR, #01, #01
 $VMSYS.JMP :?Start_stop.MAIN?EAND0010
        :?Start_stop.MAIN?AND000A
 $VMSYS.MCD MOTOR, #01, #00
```

c)
```
:0024| 0920 0300 0000 0200
:002c| 1C02 0300 4800
:0032| 0510 0400 0100
:0038| 1C02 0400 4800
:003e| 1C15 0200 01 01
:0044| 1C00 4E00
:0048| 1C15 0200 01 00
```

d)
```
:00000034| 0920 03000000 00000000 02000000
:00000042| 1C02 03000000 6E000000
:0000004c| 0510 04000000 01000000
:00000056| 1C02 04000000 6E000000
:00000060| 1C15 02000000 01 01
:00000068| 1C00 76000000
:0000006e| 1C15 02000000 01 00
```

Rys. 2. Przygotowanie programu a) kod w języku ST, b) kod w VMASM, c) kod pośredni dla maszyny 16-bitowej, d) kod pośredni dla maszyny 32-bitowej
Fig. 2. Preparation of a program a) code in ST, b) code in VMASM. c) intermediate code for 16-bit machine, d) intermediate code for 32-bit machine

co postawiło pod znakiem zapytania jej użyteczność. Szczególnie dotyczyło to rozwiązań opartych o architekturę ARM, w tym układów STM32. Zauważono również, że wydajność ta różni się w zależności od konkretnego modelu.

3.2. Architektura mikrokontrolerów STM32

Architektura ARM wykorzystywana w rodzinie STM32 charakteryzuje się pracą na adresacji wyrównanej [4, 5]. Dostęp wyrównany rozumie się jako operacja odczytu, w której adres danych jest podzielny przez liczbę bajtów określających ich rozmiar. W przypadku słowa dotyczy to podzielności adresu przez cztery, a dla półsłowa podzielności adresu przez dwa. Dostęp do bajtów jest zawsze wyrównany. Zależnie od wersji architektury niektóre operacje działania na adresach nieparzystych oraz tzw. półsłowach (*half-word*) są niedozwolone.

Jako platformy testowe do badań wykorzystano mikrokontrolery STM32, które wyposażone są procesory ARM różnych generacji. Seria mikrokontrolerów w podstawowym zakresie obejmuje następujące rdzenie:
- Cortex M0/M0+ (ARMv6-M, Von Neumann),
- Cortex M3 (ARMv7-M, Harvard),
- Cortex M4 (ARMv7E-M, Harvard),
- Cortex M7 (ARMv7E-M, Harvard).

Ze względu na duże różnice między ARMv6-M oraz ARMv7-M z perspektywy działania maszyny wirtualnej zdecydowano się na użycie serii STM32F0 oraz STM32F4 [4, 5]. Wykorzystane mikrokontrolery to:
- STM32F072RB (Cortex-M0),
- STM32F446RB (Cortex-M4).

Na podstawie specyfikacji wybranych rdzeni ARM określono istotne dane dotyczące wsparcia instrukcji procesora dla pracy z niewyrównanymi danymi w pamięci. Najniższa seria F0 wykorzystująca rdzeń Cortex-M0 nie ma wsparcia dla operacji na niewyrównanych danych, a każdorazowa próba wywołania takiej czynności na pamięci powoduje krytyczny błąd procesora. W przypadku wyższej serii Cortex-M4 ze zwiększoną liczbą instrukcji, otrzymujemy wsparcie dla pracy z dostępem do niewyrównanych danych. Dostęp dotyczy operacji odczytu i zapisu takich jak LDR, LDRH, STR, STRH.

3.3. Testy wydajnościowe

Pierwotna implementacja maszyny wirtualnej nie uwzględniała problemu wyrównania danych w architekturze ARM. Efektem tego było występowanie zabronionych operacji na pamięci, które skutkowały krytycznym błędem pracy procesora (*Hard Fault*). Był on związany z dostępem do niewyrównanych adresów w pamięci. Wprowadzono więc rozwiązanie wykorzystujące bajtowy dostęp do pamięci (zob. niżej), który wyeliminował te sytuacje, lecz okazał się niekorzystny wydajnościowo, szczególnie w maszynie 32-bitowej. Opracowano więc kolejne rozwiązania przedstawione w dalszej części pracy, oparte o kopiowanie pamięci i wyrównanie danych. W celu klasyfikacji jakości tych rozwiązań zaprojektowany został zestaw testów mających na celu ocenę szybkości oraz poprawności działania maszyny wirtualnej. Wykonanie serii testów pozwala na przebadanie wydajności zastosowanego rozwiązania sprzętowego i określenie właściwej metody dostępu do danych. Programy testowe realizowały:
- obliczanie liczb doskonałych,
- poszukiwanie liczb pierwszych,
- konwersja liczb w systemie dziesiętnym na binarny,
- symulator statku [16].

W dalszej części pracy programy te określane są jako *Test1-3* oraz *ShipSim* (Tabela 1). Pierwszy wykorzystuje pętle oraz operację *modulo*. Drugi wprowadza dodatkowo operacje na tablicach. Trzeci realizuje operacje bitowe i przesunięcia. Symulator statku jest dużym programem wykonującym działania na zmiennych typu REAL.

Tabela 1. Opis rodzajów testów oraz wykorzystanych operacji maszyny wirtualnej
Table 1. Description of types of tests and virtual machine operations used

Test1	Liczby doskonałe	FOR, WHILE, MOD
Test2	Liczby pierwsze	WHILE, MOD, TABLICE
Test3	Konwersja liczb (system 10 → 2)	FOR, AND, SHL, SRH, OR, XOR
ShipSim	Symulator statku	REAL

W dalszej części pracy przedstawiono czasy wykonywania tych testów wyrażone w milisekundach. Ze względu na dużą różnice w czasie wykonywania programu wyniki dla symulatora okrętu (*ShipSim*) zostały zwiększone 500-krotnie dla łatwiejszego porównania.

4. Metoda BYTE_ACCESS

4.1. Charakterystyka rozwiązania

W celu ominięcia ograniczenia dostępu do danych i uzyskania kompatybilności maszyny wirtualnej z architekturą ARM opracowana została metoda bajtowego dostępu do pamięci (BYTE_ACCESS). Polega ona na składaniu dowolnego typu danych (2-, 4-, 8-bajtowych) z pojedynczo odczytanych bajtów. Pobranie półsłowa (16 bitów) składa się z dwóch operacji odczytu bajtu danych oraz jednej sumy bitowej. Odczyt słowa (32-bitowego) polegał na odczytaniu dwóch półsłów, a następnie złożenia danych za pomocą sumy. Na rys. 3 przedstawiono implementację instrukcji pobierania danych programu w maszynie wirtualnej za pomocą metody BYTE_ACCESS.

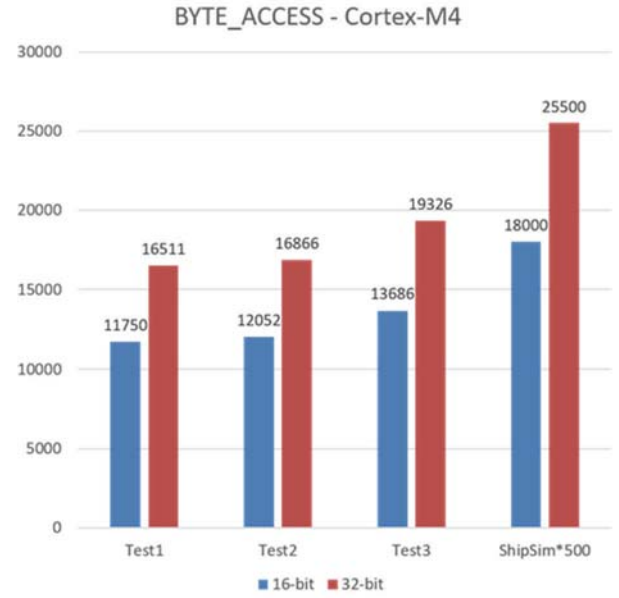
Metoda ta pozwoliła na poprawny odczyt pamięci uniezależniając go od ograniczeń występujących w niektórych rdzeniach ARM (na przykład Cortex-M0). Wadą tego rozwiązania jest zwiększony czas potrzebny na składanie półsłowa (2 bajty), a szczególnie pełnych słów danych (4 bajty).

```
16-bit: GetProgramWord
  •   WM_WORD lo = GetProgramByte();
  •   WM_WORD hi = GetProgramByte();
  •   return (hi<<8) | lo;
32-bit: GetProgramDWord
  •   WM_DWORD lo = GetProgramWord();
  •   WM_DWORD hi = GetProgramWord();
  •   return (hi<<16) | lo;
```

Rys. 3. Implementacja instrukcji pobierania zmiennych typu WM_WORD oraz WM_DWORD w trybie BYTE_ACCESS
Fig. 3. Implementation of the instruction for load variables of types WM_WORD and WM_DWORD in BYTE_ACCESS mode

Warto zauważyć, że niektóre wersje architektury ARM (np. ARMv7) wspierają część instrukcji ładowania i zapisu z niewyrównanych adresów. W ich przypadku maszyna wirtualna w wersji 16- i 32-bitowej pracuje poprawnie nie generując krytycznego wyjątku (Hard Fault). Należy jednak zwrócić uwagę, że sprzętowe wsparcie dla pracy z niewyrównanymi adresami jest często obarczone zwiększoną liczbą cykli procesora na wykonanie tych instrukcji.

Wykres na rysunku 4 przedstawia porównanie trybu 16- i 32-bitowego z trybem pracy BYTE_ACCESS na wydajniejszej jednostce z Cortex-M4. Ze względu na brak wykorzystania przez BYTE_ACCESS dodatkowych sprzętowych rozwiązań, które ma nowsza architektura, różnica między wersją 32- i 16-bitową jest porównywalna z wynikami dla Cortex-M0. Nadal 32-bitowa wersja maszyny wirtualnej jest nawet o 40 % wolniejsza.



Rys. 4. Wykres przedstawiający czas wykonywania pojedynczego cyklu programu testowego maszyny wirtualnej 16- i 32-bitowej dla metody BYTE_ACCESS na mikrokontrolerze z rdzeniem Cortex-M4
Fig. 4. Graph showing the execution time of a single cycle of the 16- and 32-bit virtual machine test program for the BYTE_ACCESS method on a microcontroller with Cortex-M4 core

5. Metoda MEMCPY_ACCESS

5.1. Charakterystyka rozwiązania

W celu zapewnienia kompatybilności z większą liczbą wersji rdzeni ARM i innych architektura oraz poprawienia wydajności pracy maszyny wirtualnej dostęp bajtowy zastąpiony został instrukcją kopiowania *memcpy* dostępną w standardowej bibliotece języka C. Zmiana powoduje przerzucenie odpowiedzialności wyboru optymalnej metody (instrukcji procesora) na kompilator. Zaletą tego rozwiązania jest wykorzystanie kompilatora, który dobiera właściwe rozwiązanie zależne od możliwości konkretnej platformy.

Funkcja *memcpy* wymaga podania wskaźnika do obiektu docelowego, źródłowego oraz rozmiaru do skopiowania – w naszym przypadku *WM_WORD* oraz *WM_DWORD*, czyli odpowiednio dwóch lub czterech bajtów danych (rys. 5).

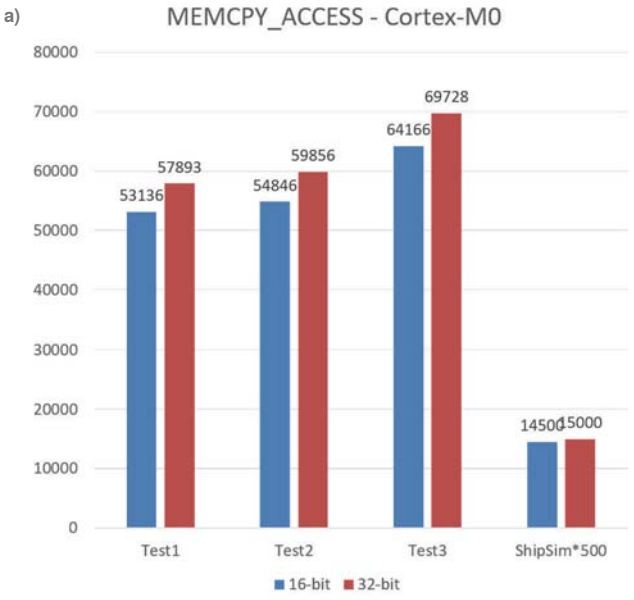
```
16-bit: GetProgramWord
  •   WM_WORD wRes;
  •   memcpy(&wRes, getCodePtr(wProgramCounter),
      sizeof(WM_WORD));
32-bit: GetProgramDWord
  •   WM_DWORD wRes;
  •   memcpy(&wRes, getCodePtr(wProgramCounter),
      sizeof(WM_DWORD));
```

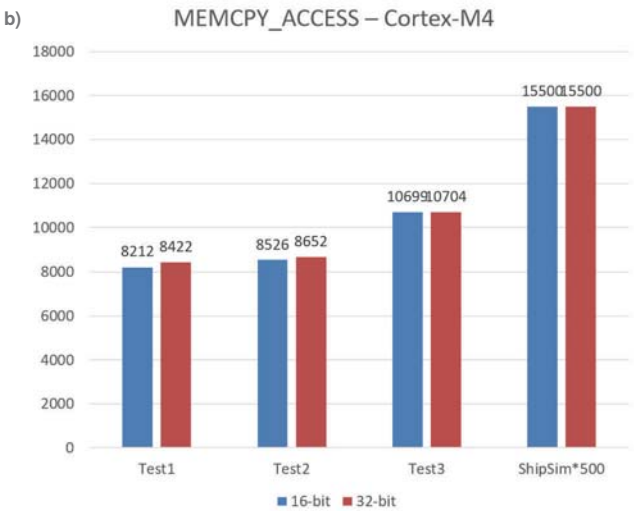
Rys. 5. Implementacja instrukcji pobierania zmiennych typu WM_WORD oraz WM_DWORD w trybie MEMCPY_ACCESS
Fig. 5. Implementation of the instruction for load variables of types WM_WORD and WM_DWORD in MEMCPY_ACCESS mode

5.2. Test wydajności

Metoda MEMCPY_ACCESS nie wymaga zmian w pozostałych modułach CPDev, w tym kompilatora oraz utworzenia nowego kodu pośredniego. Nie powoduje też nieoczekiwanych błędów spowodowanych niewspieraną metodą odczytu danych przez wykorzystaną architekturę.

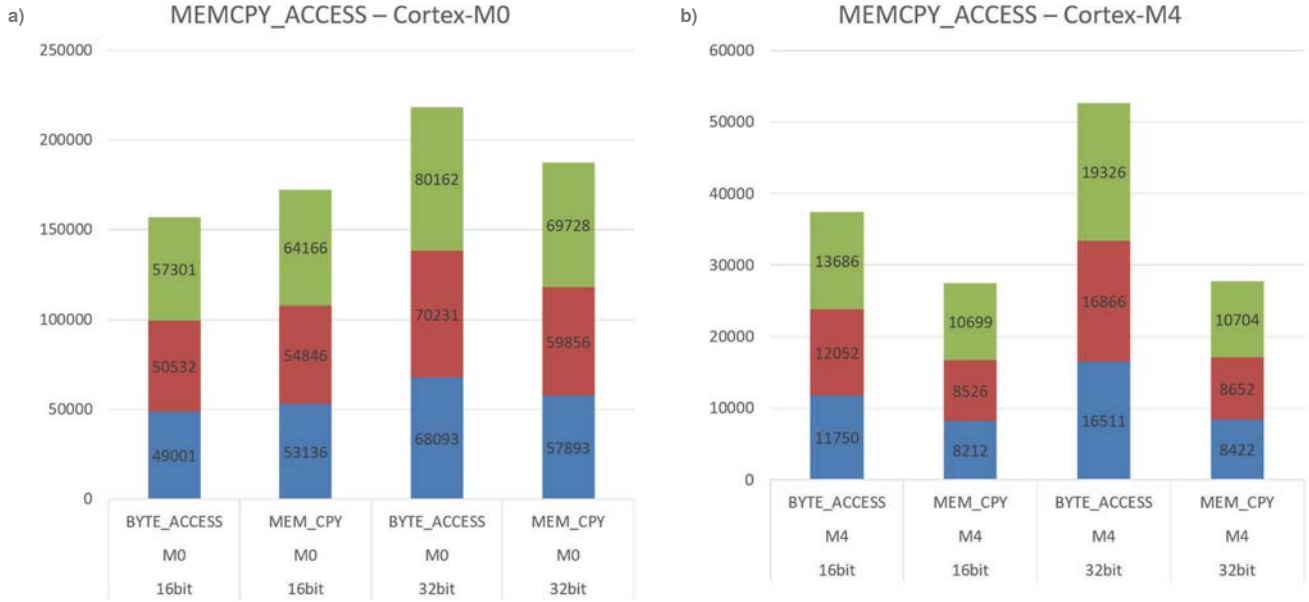
Zaproponowane rozwiązanie przetestowane zostało w tych samych warunkach co BYTE_ACCESS. Wyniki poszczególnych testów dla Cortex M0 i M4 przedstawiono na rys. 6a i 6b. Efektem rozwiązania było w większości przypadków zauważalne zwiększenie wydajności pracy maszyny wirtualnej. Na podstawie pierwszych obserwacji w obrębie jednej platformy można zauważyć zmniejszenie różnicy czasu wykonywania zadania między 16- i 32-bitową wersją maszyny wirtualnej. Przy metodzie *BYTE_ACCESS* uzyskaliśmy spadek wydajności przy tej drugiej na poziomie 25–40 %, tutaj oscyluje w granicy 3–9 %. W przypadku Cortex-M4 wersja 32-bitowa tylko w drobnej mierze cechuje się pogorszoną wydajnością (*Test1* i *Test2*). Dzięki zastosowaniu *memcpy* otrzymaliśmy tu praktycznie brak wpływu zastosowanej wersji maszyny wirtualnej, a wybór wersji 32-bitowej nie pociąga za sobą odczuwalnego spowolnienia pracy systemu.





Rys. 6. Czasy wykonywania pojedynczego cyklu programu testowego maszyny wirtualnej 16- i 32-bitowej dla metody MEMCPY_ACCESS na mikrokontrolerze z rdzeniem Cortex-M0 (a) i Cortex-M4 (b)
Fig. 6. Execution times of a single cycle of the 16- and 32-bit virtual machine test program for the BYTE_ACCESS method on a microcontroller with the cores Cortex-M0 (a) and Cortex-M4 (b)

Na rysunku 7 przedstawiono porównanie wyników uzyskanych w metodzie MEMCPY_ACCESS względem BYTE_ACCESS. Nowa metoda pozytywnie wpłynęła na praktycznie wszystkie wyniki czasu obliczeń. Warto jednak zauważyć, że na platformie Cortex-M0 były one o kilka procent gorsze niż w trybie BYTE_ACCESS dla wersji 16-bitowej.



Rys. 7. Porównanie czasów wykonywania pojedynczego cyklu programu testowego maszyny wirtualnej 16- i 32-bitowej dla metody BYTE_ACCESS oraz MEMCPY_ACCESS na mikrokontrolerze z rdzeniem Cortex-M0 (a) i Cortex-M4 (b)

Fig. 7. Comparison of execution times of a single cycle of the 16- and 32-bit virtual machine test program for the BYTE_ACCESS and MEMCPY_ACCESS method on a microcontroller with the cores Cortex-M0 (a) and Cortex-M4 (b)

6. Metoda ALIGN_4B – wyrównanie danych

6.1. Charakterystyka rozwiązania

Ze względu na brak wsparcia przez niektóre układy mikroprocesorowe odczytu danych wyrównanych do pełnych słów zdecydowano się na opracowanie nowego kompilatora CPDev, którego zadaniem jest tworzenie kodu pośredniego uwzględniającego takie ograniczenia. Wprowadzony został dodatkowy narzut informacji mający na celu generowanie instrukcji wyłącznie w odstępach czterobajtowych. Rozwiązanie to nakierowane jest ściśle na obsługę przez procesory z adresacją 32-bitową, w szczególności z architekturą ARM.

Do tej pory kompilator generował kod niewyrównany zarówno w wersji 16- jak i 32-bitowej. Część instrukcji mogła zawierać niewyrównaną liczbę słów, co powodowało dowolność w adresach kolejnych instrukcji. Widać to dobrze na rys. 2d, gdzie instrukcja inicjująca MCD ma niewyrównaną liczbę bajtów operandów. W celu przyspieszenia wykonywania instrukcji w architekturze 32-bitowej zastosowano wyrównanie kodu instrukcji i jej operandów do granicy czterech bajtów.

W celu odróżnienia instrukcji niewyrównanych od ich wyrównanych odpowiedników wprowadzono w pliku opisującym platformę docelową dodatkowe kody instrukcji. I tak zamiast instrukcji inicjującej MCD używana jest instrukcja MCD4A, FPAT wypełniająca obszar danych zostaje zastąpiona przez FPAT4A, a odczytująca wartości spod wskaźników DPRDL – przez DPRDL4A. W zastosowanym podejściu funkcje te mają dodatkowo większe możliwości niż ich wcześniejsze odpowiedniki. Przykładowo, stosowanie MCD4A nie ma ograniczenia maksymalnej długości inicjowanego bloku do 255 B. Funkcje te zostały zaimplementowane w maszynie wirtualnej jako opcja dostępna po zdefiniowaniu symbolu kompilacji ALIGN_4B.

Wprowadzenie nowych wyrównanych instrukcji odbiło się jednak negatywnie na wcześniej skompilowanych bibliotekach dostarczanych wraz ze środowiskiem CPDev. Wraz z pojawieniem się różnych instrukcji inicjujących konieczne było wprowadzenie nowego formatu bibliotek prekompilowanych, które zawierałyby obie prekompilowane postacie biblioteki (z wyrównaniem i bez). Jest to rozwiązanie znane choćby z formatu wykonywalnego Mach-O stosowanego na platformie Apple [17]. Linker sprawdza, jaką architekturę aktualnie kompiluje i do wynikowego kodu dołącza tylko tę właściwą postać biblioteki.

6.2. Test wydajności

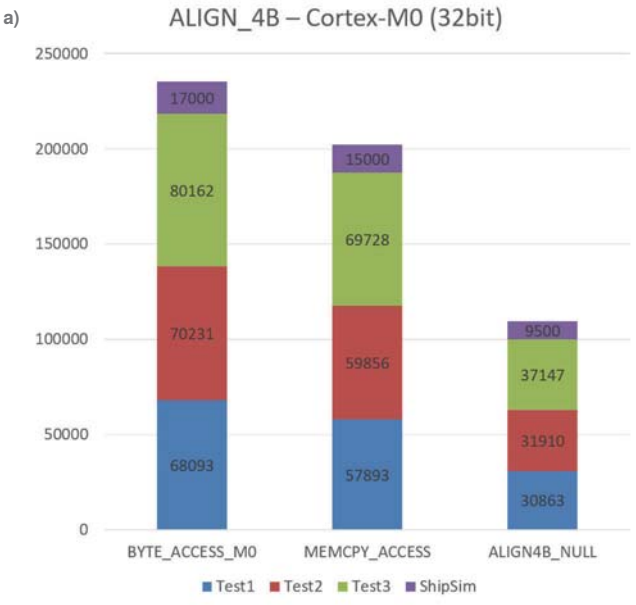
Nowy tryb pracy ALIGN_4B nakierowany jest na zastosowanie w architekturze niewspierającej dostępu niewyrównanego, a zarazem charakteryzującej się efektywną pracą z 32-bitowymi danymi. Wyniki dla Cortex-M0 czyli mikrokontrolera niewspierającego pracy oraz Cortex-M4, który go wspiera, przedstawiono na rys. 8. W każdym z testów można zauważyć pozytywny wpływ ALIGN_4B na wydajność dla maszyny w wersji 32-bitowej.

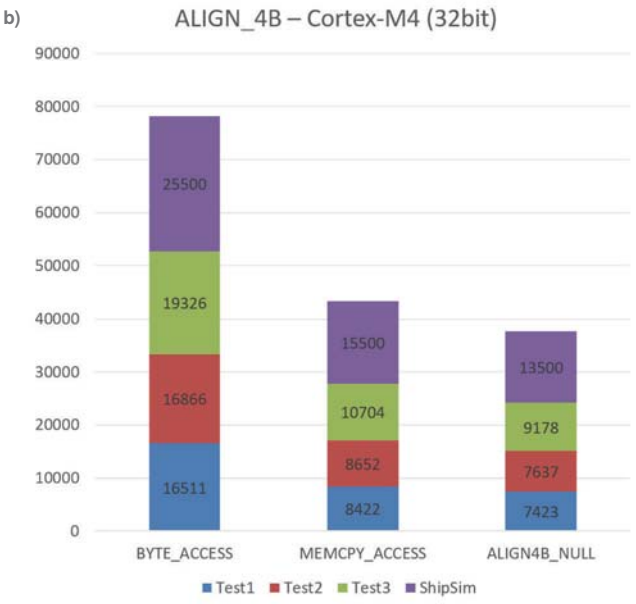
Dla Cortex-M4 różnica między MEMCPY_ACCESS oraz ALIGN_4B jest mniejsza niż dla Cortex-M0, ale nadal pozwala na zmniejszenie czasu wykonywania zadań. W obydwu przypadkach ALIGN_4B, w porównaniu do BYTE_ACCESS, pozwala na zmniejszenie czasu cyklu prawie dwukrotnie.

6.3. Rozmiar kodu wynikowego

Wadą tej metody ALIGN_4B jest zwiększony rozmiar kodu wynikowego. Dla przykładu instrukcja inicjująca 1 bajt bez wyrównania na architekturze 32-bitowej zajmuje 8 bajtów (2 B – kod instrukcji, 4 B – adres, 1 B – długość danych, 1 B – dane). W wersji z wyrównaniem zajmie ona 16 bajtów (każdy operand 4 B). Tak pesymistyczny przypadek zdarza się nieczęsto, jednakże liczba bajtów podczas inicjowania przyrasta skokowo w odróżnieniu od przyrostu liniowego występującego w wersji bez wyrównania. Rysunek 9 zawiera kod pośredni wygenerowany przez kompilator pracujący w trybie ALIGN_4B i wykorzystujący instrukcję MCD4A o kodzie 1C370000 zamiast MCD.

Na rysunku 10 przedstawiono porównanie rozmiaru kodu programu ShipSim w zależności od wersji maszyny (16-, 32-bitowa, 32-bitowa z wyrównaniem). Kod 32-bitowy jest o około 50 % większy od 16-bitowego, natomiast wyrównanie pociąga za sobą praktycznie dwukrotny wzrost kodu wynikowego względem 16-bitowej wersji.





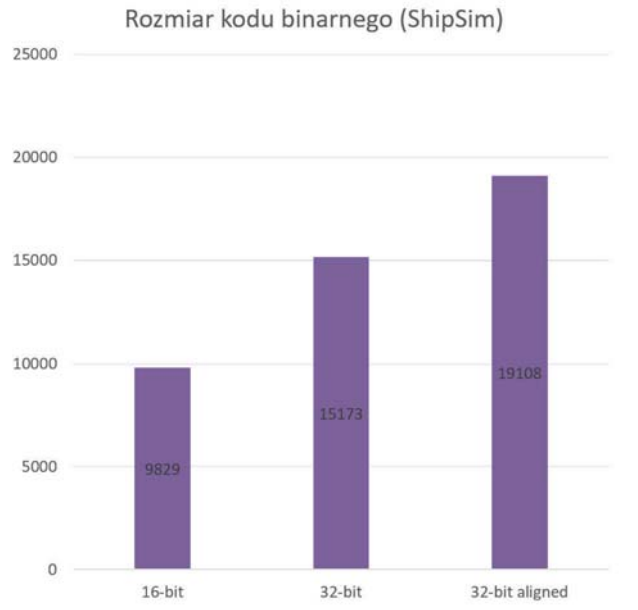
Rys. 8. Wykres przedstawiający porównanie czasów wykonywania pojedynczego cyklu programu testowego maszyny wirtualnej 32-bitowej dla metody BYTE_ACCESS, MEMCPY_ACCESS, oraz ALIGN4B_NULL na mikrokontrolerze z rdzeniem Cortex-M0 (a) i Cortx-M4 (b)

Fig. 8. Graph showing the comparison of the execution times of a single cycle of the 32-bit virtual machine test program for the BYTE_ACCESS, MEMCPY_ACCESS, and ALIGN4B_NULL methods on a microcontroller the cores Cortex-M0 (a) and Cortex-M4 (b)

```
:00000034| 09200000 03000000 00000000 02000000
:00000044| 1C020000 03000000 6E000000
:00000050| 05100000 04000000 01000000
:0000005C| 1C020000 04000000 6E000000
:00000068| 1C370000 02000000 01000000 01000000
:00000078| 1C000000 76000000
:00000080| 1C370000 02000000 01000000 00000000
```

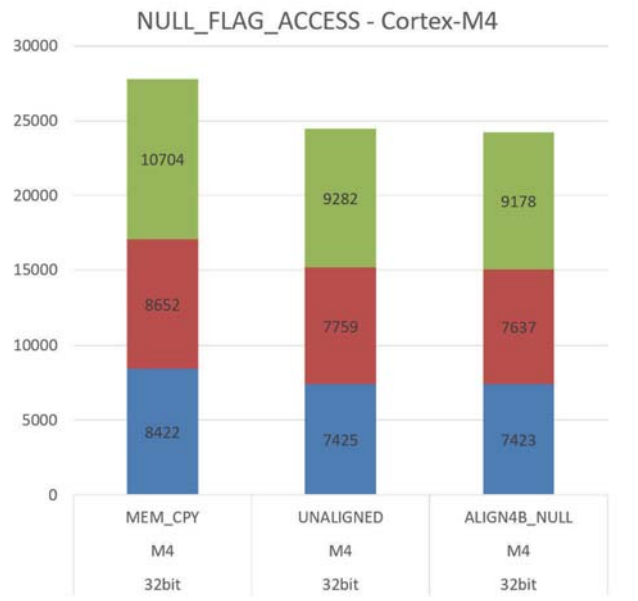
Rys. 9. Kod pośredni dla maszyny 32-bitowej z wyrównaniem danych (por. rys. 2d)

Fig. 9. Intermediate code for a 32-bit machine with data alignment (see fig. 2d)



Rys. 10. Wykres przedstawiający porównanie objętości kodu maszyny wirtualnej 16-bitowej, 32-bitowej oraz 32-bitowej z wyrównaniem do czterech bajtów

Fig. 10. Graph showing the comparison of code size virtual machine 16-bit, 32-bit and 32-bit aligned for four bytes



Rys. 11. Wykres przedstawiający porównanie czasów wykonywania pojedynczego cyklu programu testowego maszyny wirtualnej 32-bitowej dla metody MEMCPY_ACCESS, UNALIGNED oraz ALIGN4B_NULL na mikrokontrolerze z rdzeniem Cortex-M4

Fig. 11. Graph showing the comparison of the execution times of a single cycle of the 32-bit virtual machine test program for the MEMCPY_ACCESS, UNALIGNED and ALIGN4B_NULL methods on a microcontroller with Cortex-M4 core

6.4. Podsumowanie

Opracowane rozwiązania pozwalają na dostosowanie maszyny do wybranej architektury, na której będą uruchamiane, zachowując przy tym jej przenaszalność. Kolejne opracowywane generacje procesorów mogą pozytywnie wpływać na wydajność rozwiązań programowych, wcześniej ograniczonych przez cechy sprzętu. Cortex-M4 z architekturą ARMv7 wspiera w pewnym stopniu pracę z niewyrównanymi danymi, w którym oryginalna wersja maszyny wirtualnej działa bez żadnych zmian. Tym

samym możliwe jest używanie maszyny wirtualnej na takim mikrokontrolerze bez wsparcia dodatkowych instrukcji składania rozkazów z mniejszych elementów lub innych modyfikacji (tryb bezpośredniego dostępu, UNALIGNED).

Rysunek 11 przedstawia porównanie 32-bitowych wersji maszyny wirtualnej pracującej w trzech trybach. Widać, że MEMCPY_ACCESS jest zauważalnie najgorszą z metod pobierania rozkazów maszyny. Wersja wyrównana do 4 B przynosi niewielki pozytywny skutek na czas wykonywania kodu maszyny wirtualnej względem wersji niewyrównanej.

W praktyce przemysłowej dominują wciąż mikrokontrolery wcześniejszych generacji. Stąd ogólne wnioski można streścić następująco. W przypadku starszych rozwiązań architektury należy używać trybu BYTE_ACCESS w wersji 16-bitowej, a MEMCPY_ACCESS w 32-bitowej. Wyrównanie (ALIGN_4B) przynosi istotny efekt wydajnościowy w trybie 32 bitowym kosztem zwiększonego rozmiaru kodu. W nowszej architekturze (ARMv7) zysk z wyrównania także istnieje, ale jest niewielki.

Podziękowania

Projekt finansowany w ramach programu Ministra Nauki i Szkolnictwa Wyższego pod nazwą „Regionalna Inicjatywa Doskonałości" w latach 2019–2022 nr projektu 027/RID/2018/19 kwota finansowania 11 999 900 zł.

Bibliografia

1. CODESYS (2017). Codesys download area, [www.codesys.com/download.html].
2. CPDev engineering environment, [https://cpdev.kia.prz.edu.pl].
3. STM32 32-bit Arm Cortex MCUs, [www.st.com/en/microcontrollers-microprocessors/stm32--32-bit-arm-cortex-mcus.html].
4. Arm Cortex-M0 (ARMv6), [https://developer.arm.com/documentation/dui0497/a].
5. Arm Cortex-M4 (ARMv7), [https://developer.arm.com/documentation/100166/0001].
6. ECMA-335, S. (2012). Common Language Infrastructure (CLI), Ecma, Geneva.
7. Hajduk Z., Trybus B., Sadolewski J., *Architecture of FPGA embedded multiprocessor programmable controller*, "IEEE Transactions on Industrial Electronics", Vol. 62, No. 5, 2015, 2952–2961, DOI: 10.1109/TIE.2014.2362888.
8. IEC 61131-3, S. (2013). *Programmable Controllers*. Part 3. Programming languages, IEC, International Standard.
9. ISaGRAF, [www.rockwellautomation.com/en-us/support/documentation/technical-data/isagraf_20190326-0743.html].
10. Lindholm T., Yellin F., Bracha G., Buckley A., *The Java Virtual Machine Specification*, Oracle America, Inc. 2013.
11. Rzońca D., Sadolewski J., Stec A., Świder Z., Trybus B., Trybus L., *Open environment for programming small controllers according to IEC 61131-3 standard*, "Scalable Computing: Practice and Experience", Vol. 10, No. 3, 2009, 325–336.
12. Rzońca D., Sadolewski J., Stec A., Świder Z., Trybus B., Trybus L., *Programming controllers in structured text language of IEC 61131-3 standard*, "Journal of Applied Computer Science", Vol. 16, No. 1, 2008, 49–67.
13. Simros M., Wollschlaeger M., Theurich S., *Programming embedded devices in IEC 61131-languages with industrial PLC tools using PLCopen XML*, Proceedings of the CONTROLO'2012 Portuguese Conference on Automatic Control, Funchal, Portugal, 51–56.
14. Trybus B., *Development and Implementation of IEC 61131-3 Virtual Machine*, "Theoretical and Applied Informatics", Vol. 23, No. 1, 2011, 21–35, DOI: 10.2478/v10179-011-0002-z.
15. Rzońca D., Sadolewski J., Stec A., Świder Z., Trybus B., Trybus L., *Developing a Multiplatform Control Environment*, "Journal of Automation, Mobile Robotics and Intelligent Systems", Vol. 13, No. 4, 2019, 73–84, DOI: 10.14313/JAMRIS/4-2019/40.
16. Rzońca D., Sadolewski J., Stec A., Świder Z., Trybus B., Trybus L., *Ship Autopilot Software – A Case Study*. In: Bartoszewicz A., Kabziński J., Kacprzyk J. (eds), Advanced, Contemporary Control. Advances in Intelligent Systems and Computing, Vol. 1196, Springer, Cham. DOI: 10.1007/978-3-030-50936-1_124.
17. Velu V.K., *Mobile Application Penetration Testing*. Packt Publishing, 2016, ISBN 9781785883378.

The Performance of Executing Intermediate Code for Control Systems Using STM32 Architecture

Abstract: The article presents performance tests of code executed by STM32 microcontrollers using a virtual machine (so-called intermediate code) dedicated to control systems. The ARM architecture used in these chips has limitations related to access to non-aligned addresses. Three ways to overcome these limitations have been proposed, and each has been subjected to a suite of tests to determine their performance. Tests were conducted for two operating modes, i.e. with 16- and 32-bit addressing for different generations of chips. The test results allow to choose the right solution for a specific platform.

Keywords: STM, ARM, virtual machine, intermediate code

mgr inż. Marcin Hubacz
m.hubacz@prz.edu.pl
ORCID: 0000-0002-2748-11454

W 2019 r. ukończył studia na Wydziale Elektrotechniki i Informatyki Politechniki Rzeszowskiej – kierunek Automatyka i Robotyka oraz Informatyka. Obecnie Asystent w Katedrze Informatyki i Automatyki Politechniki Rzeszowskiej. Jego główne zainteresowania dotyczą robotyki, elektroniki, systemów wbudowanych oraz druku 3D.

dr inż. Jan Sadolewski
jsad@prz.edu.pl
ORCID: 0000-0001-7370-9027

Absolwent Wydziału Elektrotechniki i Informatyki Politechniki Rzeszowskiej (2006 r.). W 2012 r. uzyskał stopień doktora nauk technicznych w dyscyplinie informatyka na Wydziale Automatyki, Elektroniki i Informatyki Politechniki Śląskiej w Gliwicach. Jego zainteresowania naukowe koncentrują się wokół języków programowania, tworzenia kompilatorów oraz środowisk wykonawczych.

dr inż. Bartosz Trybus
btrybus@kia.prz.edu.pl
ORCID: 0000-0002-4588-3973

Adiunkt w Katedrze Informatyki i Automatyki Politechniki Rzeszowskiej. Ukończył studia na Wydziale Elektrycznym, Automatyki, Informatyki i Elektroniki AGH w Krakowie. Doktorat z informatyki uzyskał w 2004 r. Jego główne badania dotyczą systemów czasu rzeczywistego i środowisk wykonawczych oprogramowania sterującego.